O'REILLY

TURING 图灵程序设计丛书



Java性能权威指南

Java Performance: The Definitive Guide

[美] Scott Oaks 著

柳飞 陆明刚 臧秀涛 译

中国工信出版集团 人民邮电出版社 POSTS & TELECOM PRESS

数字版权声明

译者介绍



柳飞

码农，毕业于中国科学技术大学和复旦大学，现就职于喜马拉雅FM，从事运营系统研发。马拉松理性爱好者，赛道急救志愿者和急救兔。中度拖延症患者。



陆明刚

毕业于四川大学，目前在EMC中国卓越研发集团任首席工程师，曾任趋势科技中国软件研发中心技术经理，在信息科学和工程领域有十余年的实践和研究经验，拥有多项中国及美国专利。关注JVM性能调优和大数据及其实践，喜欢挖掘技术背后的内幕并乐此不疲。



臧秀涛

中国科学院计算技术研究所硕士，对编程语言、虚拟机等领域有浓厚兴趣。先后从事过游戏服务器、操作系统方面的开发，现于InfoQ任QCon大会主编，推动软件开发领域内实践经验的传播。业余喜欢读书、翻译，有《C++ API设计》《Groovy程序设计》等译作。期待与读者通过微博（@臧秀涛）或微信公众号dev-news联系。

TURING 图灵程序设计丛书

Java 性能权威指南

Java Performance
The Definitive Guide

[美] Scott Oaks 著
柳飞　陆明刚　臧秀涛 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo
O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　京

图书在版编目（C I P）数据

Java性能权威指南 /（美）奥克斯（Oaks,S.）著；
柳飞，陆明刚，臧秀涛译. -- 北京 : 人民邮电出版社，
2016.3
（图灵程序设计丛书）
ISBN 978-7-115-41376-5

Ⅰ. ①J… Ⅱ. ①奥… ②柳… ③陆… ④臧… Ⅲ. ①
JAVA语言—程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2016)第003957号

内 容 提 要

本书对 Java 7 和 Java 8 中影响性能的因素展开了全面深入的介绍，讲解传统上影响应用性能的 JVM 特征，包括即时编译器、垃圾收集、语言特征等。内容包括：用 G1 垃圾收集器最大化应用的吞吐量；使用 Java 飞行记录器查看性能细节，而不必借助专业的分析工具；堆内存与原生内存最佳实践；线程与同步的性能，以及数据库性能最佳实践等。

本书适合需要了解 JVM 调优的 Java 新手以及想要最优化应用性能的成熟开发人员阅读。

◆ 著　　　　[美] Scott Oaks
　译　　　　柳　飞　陆明刚　臧秀涛
　责任编辑　岳新欣
　执行编辑　张　曼
　责任印制　杨林杰

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
　邮编　100164　　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京　　　　印刷

◆ 开本：800×1000　1/16
　印张：20.5
　字数：487千字　　　　2016年 3 月第 1 版
　印数：1 – 4 000册　　　2016年 3 月北京第 1 次印刷

著作权合同登记号　图字：01-2014-6317号

定价：79.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第 0021 号

版权声明

O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——*Wired*

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

推荐序

可能很多 Java 程序员都会觉得解决性能问题是一件特别苦恼、特别让人抓狂的事情，因为系统的各个层面的问题都会导致性能问题。JVM 优化是个老生常谈的话题，也是程序员面试容易遇到的高频问题，貌似每个程序员或者面试官都知道那么一点儿，没有什么新意。市面上 JVM 性能方面的书也有几本，但是真正把这些知识整理成书，能够做到紧跟时代步伐的并不多。很多资料都忽略了 JVM 最近几年的发展和进步，这样我们就无法发挥 JVM 那些最激动人心的新特性，也会使自己像一个生活在 21 世纪的原始人。很多原来的优化方法都已失去了存在的意义，但还是被大家不断地讨论。片面、零散、落伍的知识在 JVM 领域大行其道，要命的是其中还有很多是错误的。当然，我们并不能怪那些多年前的作者，希望他们能够预知今天 JVM 的进展。但如果想再找一本能跟得上时代步伐的 JVM 调优的书的话，貌似当下只有这本书。这就是我推荐此书的理由：全面、实用、紧跟时代。本书很多章节都是我非常喜欢的，比如关于 JMC 的。相信很多有多年 JVM 调优经验的人也未必听说过 JMC，但不得不说，每个遇见 JMC 的人都如获至宝。

本书完整地介绍了 JVM 调优需要的方方面面，而不仅仅限于那些诡异参数的简单说明，非常具有实用性和系统性。值得一提的是，几位译者都是在这个领域非常资深的专家，翻译水平上乘。我觉得每个对 JVM 感兴趣的程序员都应该看看这本书。

程显峰

Kage 技术咨询合伙人

这是一部关于 Java 性能调优的卓越作品。该书涉及性能测试、性能分析、性能调优的原理、方法、工具等诸多方面，书中最新的 JVM 和体系结构的相关知识可以帮助我们更好地理解 Java，同时该书又包含了许多非常工程性的经验，比如多线程、数据库、序列化以及 Java API 等，这些经验不仅对 Java 工程师很有帮助，也为其他开发人员及性能调优人员提供了问题解决思路和方法上的启迪。借助这本书，我们可以从 Java 纷繁复杂的性能调优参数中解脱出来，看到背后的动机和缘由，从而获得对性能的不一样的理解。

邹飞

Google 资深软件工程师，技术经理

前言

起初 O'Reilly 公司让我写一本关于 Java 性能调优的书时，我还不确定是否值得写。我在想，难道 Java 性能调优我们做得还不够吗？事实上，虽然我日常的基本工作是 Java（和其他）应用程序的性能调优，但我宁愿将大多数时间都花在提高应用程序的算法效率以及处理外部系统的性能瓶颈上，而不是直接进行 Java 自身性能的调优。

但转念一想，我不禁哑然失笑（像往常一样）。的确，我的大量时间都花在了端到端的系统性能调优上，有时会发现那些原本可以用 $O(\log N)$ 却用了 $O(n^2)$ 算法的代码。不过这也说明，我每天考虑的都是 GC 调优、JVM 编译器的性能调优，或者是如何使 Java EE API 的性能发挥到极致。

说这些并不是想要抹杀过去 15 年里 Java 和 JVM 在性能上取得的巨大进步。1990 年代晚期，我在 Sun 公司担当 Java 布道师，当时仅有的真正意义上的“基准测试”工具来自 Pendragon 软件的 CaffeineMark 2.0。由于种种原因，该基准测试工具设计上的不足很快就限制了它的价值；然而在那个年代，我们总喜欢告诉所有人，依据这个基准测试，Java 1.1.8 的性能比 Java 1.0 快八倍。这并非耸人听闻——Java 1.1.8 已经有了真正的即时编译器，而 Java 1.0 差不多完全是解释型的。

之后，Java 标准委员会开始制定更严谨的基准测试，Java 的性能测试开始围绕这些基准测试展开。最终，JVM 的所有领域——垃圾收集、编译器和 API 都获得了长足的进步。这个过程一直延续到今天，而关于性能的一个重要事实是，调优正变得越来越艰难。引入即时编译器后所获得的八倍性能提升，只是一个简单的工程问题，即使编译器持续改进，我们也无法再次看到如此巨大的改进了。并行化垃圾收集也曾极大地提升过性能，但最近的变化则是渐进式的。

这是典型的应用发展过程（JVM 本身也是另外一个应用）：在项目初期，很容易找到架构上的改进点（或代码缺陷），一旦找到就能极大改善性能。而在成熟应用中，要找到这样的性能改进点则很罕见。

起初我觉得，从很大程度上说，Java 性能调优都已经工程化了，但有几件事情让我相信我错了。首先，我每天在特定环境下运行 JVM 时，都会遇到许多这样或那样的问题。新工程师源源不断地进入 Java 领域。在特定的领域，JVM 的行为仍然相当复杂，因此有份描

述它如何运作的指南很有必要。其次，现在的计算环境发生了变化，已经影响到了工程师们所面临的性能问题。

在过去几年中，性能关注点有了分歧。一方面，有大量内存堆可运行 JVM 的大机器现在已经很普遍了。为了应对这样的变化，JVM 也有了新的垃圾收集器（G1）——这项新技术相比传统的收集器更需要手工调优。同时，云计算又提升了单 CPU 的小机器的重要性：你可以从 Oracle、Amazon 或其他公司以非常便宜的价格租用单 CPU 机器，运行小的应用服务器。（你获得的并不是真的单 CPU 机器，而是一台巨大机器上的一个虚拟 OS 镜像，但虚拟 OS 被限制为使用单个 CPU。从 Java 角度看，它和单 CPU 的机器相同。）在这些环境中，正确管理小量内存变得非常重要。

Java 平台也在持续演变。Java 的每个新版本都会提供新的语言特性和新的 API，这些特性和 API 并不总是为了提高应用性能，也是为了改善开发人员的生产率。语言特性运用得好，应用的运行就会变得轻快，反之则缓慢笨重。另外，平台的演化也带来了一些重要的性能课题：毫无疑问，程序间用 JSON 交换信息要比用高度优化的私有协议更容易。节约开发人员的时间就是巨大的收益——但真正的目的是确保生产率提升的同时，性能也能提升（至少是两者之间取得平衡）。

读者对象

本书适合那些渴望深入了解 JVM 和 Java API 性能各个方面的性能调优工程师和开发者。

假如你想快速修复性能问题，比如网站周一早上要上线，而现在已经是周日深夜了，那么本书可能不适合你。

如果你是性能分析的新手，正要开始进行 Java 的性能分析，那么本书会对你有所帮助。我的目的主要是为新工程师提供足够多的信息和上下文，以便使他们明白如何将基本的性能调优原则运用到 Java 应用中去。然而，系统分析的范畴非常广阔，已经有大量的优秀资源（那些原则当然也适用于 Java），从这个意义上来说，希望本书也能成为它们的有益补充。

不过从根本上说，想让 Java 运行得飞快，就得深入理解 JVM（以及 Java API）的实际工作原理。有数以百计的 Java 性能调优参数，而 JVM 调优并不像瞎猫碰死耗子那样，调一下再看看是否奏效。与之相反，我的目的是提供更为详尽的 JVM 和 API 工作原理，以期在你理解它们如何工作的原理之后，能理解应用的某些行为为何糟糕。理解了这些之后，排除那些性能低下、令人不快的行为就会变成简单（至少是比以前更简单）的任务。

Java 性能调优工作还有一个有趣的方面，就是开发人员的背景和性能调优或 QA 组人员的背景常常有很大差别。我认识一些开发人员，他们可以记住成千上万个令人费解的很少使用的 Java API 方法签名，但他们对 `-Xmn` 的含义却没有什么概念。我也认识一些测试工程师，他们可以通过设置垃圾收集器的各种标志来榨取最后一滴性能，但他们却很少有人能用 Java 写出像样的“Hello, World”。

Java 性能调优覆盖这两个领域：编译器和垃圾收集器等的调优参数，以及 API 的最佳实践。所以，我假定你对如何编写 Java 程序有很好的理解。即便你主要的兴趣不是在 Java 编程，我仍然会花一点时间讨论编程，包括例子中包含大量数据的示例程序。

然而，如果你的主要兴趣是 JVM 自身的性能调优——意思是不用更改任何代码而改变 JVM 的行为，那么本书的大量章节都对你有用。可以随意跳过代码部分，而关注你所感兴趣的领域。也许你会顺便为 Java 应用如何影响 JVM 性能提出一些真知灼见，并向开发人员提出更改建议，以便让你的性能调优测试工作更加如鱼得水。

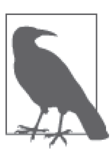
排版约定

本书使用的排版约定如下。

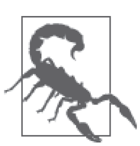
- 楷体
  表示新术语。
- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。
- 等宽粗体（**`constant width bold`**）
  表示应该由用户输入的命令或其他文本。
- 等宽斜体（*`constant width italic`*）
  表示应该由用户输入的值或根据上下文确定的值替换的文本。



这个图标代表提示或建议。



这个图标代表重要说明。



这个图标代表警告或提醒。

使用代码示例

可以在这里下载本书随附的资料（代码示例、练习题等）：https://github.com/ScottOaks/JavaPerformanceTuning。

让本书助你一臂之力。也许你需要在自己的程序或文档中用到本书中的代码。除非大段大段地使用，否则不必与我们联系取得授权。例如，无需请求许可，就可以用本书中的几段代码写成一个程序。但是销售或者发布 O'Reilly 图书中代码的光盘则必须事先获得授权。引用书中的代码来回答问题也无需授权。将大段的示例代码整合到你自己的产品文档中则必须经过许可。

使用我们的代码时，希望你能标明它的出处，但不强求。出处一般包括书名、作者、出版商和 ISBN，例如：*Java Performance: The Definitive Guide* by Scott Oaks (O'Reilly, 2014). Copyright by Scott Oaks, 978-1-449-35845-7。

如果还有关于使用代码的未尽事宜，可以随时与我们联系：permissions@oreilly.com。

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应需而变的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。

Safari Books Online 是技术专家、软件开发人员、Web 设计师、商务人士和创意人士开展调研、解决问题、学习和认证培训的第一手资料。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

> O'Reilly Media, Inc.
> 1005 Gravenstein Highway North
> Sebastopol, CA 95472

中国：

> 北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
> 奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://oreil.ly/java-performance-tdg。

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

感谢在我写这本书时所有帮助过我的人。从各方面来看，这本书汇集了我过去 15 年来在 Sun 公司和 Oracle 公司的 Java 性能调优小组中的所学所知。如果将为这本书提出真知灼见的人都列出来的话，会是一份长长的列表。感谢在那段时间与我一同工作的工程师们，特别是在那些年里耐心回答我铺天盖地的问题的人们。

我要特别感谢 Stanley Guan、Azeem Jiva、Kim LiChong、Deep Singh、Martijn Verburg 和 Edward Yue Shung Wong，感谢他们拨冗审阅本书的初稿，感谢他们提供的宝贵意见。虽然本书因为他们的建议而完善了许多，但我相信错误仍然在所难免。

O’Reilly 的工作人员总是那么乐于助人，感谢我的编辑 Meg Blanchette，感谢你在整个写作过程中的鼓励。最后，衷心感谢我的丈夫 James 给予我在为书而抓狂的漫漫长夜里的悉心陪伴，以及周末晚餐。

第1章

导论

这是一本关于 Java 性能调优科学和艺术的书。

说性能调优是门科学，这并不令人意外；性能调优涉及大量数字、检测和分析工作。性能调优工程师大多具有科学背景，只有基于严谨的科学理论才能将性能发挥到极致。

那它的艺术性呢？其实性能调优是科学与艺术的结合体这一说法并不新鲜，但我们探讨性能时却很少能清楚地意识到这一点。从某种程度上说，这可能是因为我们所受的教育训练并不容易产生“艺术”的思想火花。

说它是艺术的部分原因是，对一些人来说，艺术从根本上就是建立在知识和经验的基础上的。据说，足够先进的技术与魔术无异，例如对于圆桌骑士而言，使用手机毫无疑问就是一种魔法。与此类似，优秀性能调优工程师的工作就像是艺术，而这艺术正是源于深厚的知识、丰富的经验和敏锐的直觉。

本书的侧重点不在于三者中的经验和直觉，而是在拓展知识的深度。日积月累，这些知识将有助于提升你的技能，有助于你成为一名优秀的 Java 性能调优工程师。本书还有助于你深入理解 Java 平台性能的各个方面。

本书涉及的知识主要分两大类。首先是如何对 Java 虚拟机（Java Virtual Machine，JVM）自身的性能进行调优，即如何通过 JVM 的配置来影响程序的各种性能指标。JVM 性能调优的过程实际上与 C++ 程序员在编译时通过测试选择编译参数，以及 PHP 码农在 php.ini 文件中选择适当变量等过程非常类似，但对于那些即便有其他语言经验的 Java 开发者来说，调优过程仍然不那么令人愉快。

其次是理解 Java 平台的特性对性能的影响。注意，此处的平台既指 Java 语言（例如线程和同步），也指 Java 标准 API（例如 XML 解析性能）。虽然 Java 语言和 Java API 完全不是一回事，但本书并不作严格区分，这两方面的内容都会涵盖。

JVM 自身的性能很大程度上取决于调优标志，而 Java 平台的性能则更多由在应用代码中采用最佳实践决定。在一个开发团队中，开发人员编写代码，性能组负责性能测试。编码和调优常常被认为是两个不同的专业领域：性能调优工程师只是竭力将 JVM 的性能发挥到极致，而开发人员只关心他们的代码逻辑是否正确。这种区分没有什么意义。任何从事 Java 相关工作的人都应该熟谙代码在 JVM 中的行为，以及如何调优才能提升性能。对专业知识的全面掌握能让你的工作更具艺术气息。

1.1 概述

首先是通览：第 2 章讲述了测试 Java 应用的通用方法，还包括 Java 基准测试中的陷阱。通过可视化的性能分析，我们可以了解应用正在做什么，所以第 3 章概要介绍了监控 Java 应用的一些工具。

接下来就是深入性能调优了，首先关注的是常见的调优主题：JIT 编译（第 4 章）和垃圾收集（第 5 章和第 6 章）。剩余各章关注的是 Java 平台各方面的最佳实践：Java 堆内存的使用（第 7 章），本地内存的使用（第 8 章），线程性能的调优（第 9 章），Java 企业版 API（第 10 章），JPA 和 JDBC（第 11 章），以及一些通用的 Java SE API 技巧（第 12 章）。

附录 A 列出了本书中所有的调优标志，并给出了解释这些标志的交叉引用章节。

1.2 平台版本约定

本书基于 Oracle HotSpot JVM 和 Java Standard Edition（Java SE）7 和 8。在发布版之间，Oracle 更新会定期发布更新版本。多数时候，更新版本只修订 bug，不会加入语言新特性或者变更关键功能，但会更改调优标志的默认值。Oracle 很可能将在本书出版之后提供更新版本，当前版本为 Java 7 update 40 和 Java 8（迄今为止，还没有 Java 8 的更新版本）[1]。如果更新版本对 JVM 的行为做了重大修订，则会命名如下：7u6（Java 7 update 6）。

本书关于 Java 企业版（Java EE）的内容是基于 Java EE 7。

虽然当前版本的 Java 构建于之前的发布版，但本书并不涉及老版本 Java 的性能调优。对这本讲述性能调优的书而言，Java 7 是一个很好的起始点，因为 Java 7 引入了大量的性能新特性和优化。其中最主要的是称为 G1 的垃圾收集（GC）算法（老版本的 Java 包含了 G1 的试验版，但直至 Java 7u4，G1 才真正可用于生产环境）。Java 7 也包括许多与性能相关的新特性和增强，可以让我们更清楚地了解 Java 应用的运转。Java 8 再接再厉，平台得到了进一步增强（例如引入了 lambda 表达式）。Java 8 自身的性能得到了巨大提升，在好几个关键性领域都大大超过了 Java 7。

JVM 还有其他实现。Oracle 自己的 JRockit JVM（支持 Java SE 6）。IBM 提供了自己的 Java 兼容实现（包括 Java 7 版本）。还有许多其他公司得到许可从而可以改进 Oracle 的 Java 技术。

注 1：现在为 Java 7 update 55，Java 8 update 5。——译者注

> **Oracle 的商业版 JVM**
>
> Java 和 JVM 都是开源的，任何想参与 Java 开发的人都可以加入项目：http://openjdk.java.net。即便你不打算参与开发，也可以从上述网站免费下载源代码。本书讨论的所有内容基本上都是基于开源版的 Java。
>
> Oracle 也有商业版的 Java，可以通过支持合同获得。它基于标准开源版本的 Java 平台，但包括了一些开源版中所没有的特性。商业版 JVM 中和性能密切相关的一个特性就是 Java 飞行记录器（Java Flight Recorder，JFR，参见 3.4.1 节）。
>
> 除非特别说明，本书所有的内容都适用于开源版的 Java。

虽然只有通过兼容性测试的平台才能使用 Java 的名称，但本书不会总是围绕兼容性展开讨论。那些调优标志尤其如此。所有的 JVM 实现都有一个或多个垃圾收集器，但每个供应商所提供的 GC 实现，其调优标志都是产品特定的。所以本书所讲的概念可适用于所有 Java 实现，但具体的调优标志和建议仅适用于 Oracle 的标准（基于 HotSpot 的）JVM。

用较早版本的 HotSpot JVM 时要注意，标志及其默认值在发布版之间可能会发生变化。本书只涵盖 Java 7（直到 7u40）和 Java 8（仅首个版本），而不是试图穷尽迄今为止的各个版本。以后的版本（例如，假定是 7u60）可能会对这些信息做少许改动。重要的变更请查阅发布说明。

从 API 层面来看，不同 JVM 实现之间的兼容性很高，即便特定类在 Oracle HotSpot Java SE（或 EE）和其他平台上的实现方式也有细微的不同。类的功能必须等价，但具体实现可以变更。所幸这些并不多见，不会对性能产生重大影响。

对于本书的剩余部分，术语 Java 和 JVM 应理解为特指 Oracle HotSpot 的实现。严格来说，“JVM 首次执行时不会进行代码编译”的说法并不正确，因为有些 Java 的实现在首次执行时会编译代码。但这种略写比到处写（和读）“Oracle HotSpot JVM……”要简便得多。

JVM调优标志

除了少数例外，JVM 主要接受两类标志：布尔标志和附带参数的标志。

布尔标志采用以下语法：`-XX:+`*`FlagName`* 表示开启，`-XX:-`*`FlagName`* 表示关闭。

附带参数的标志采用以下语法：`-XX:`*`FlagName=something`*，表示将标志 `flagName` 的值设置为 `something`。其中 `something` 通常可以为任意值。例如 `-XX:NewRatio=`*`N`*，表示 `NewRatio` 可以设置为任意值 `N`（`N` 是我们讨论所关注的重点）。

介绍每个标志时，我们会讨论它的默认值。默认值的选取通常综合考虑了不同因素：运行 JVM 的物理平台，以及其他传给 JVM 的命令行参数。如有疑问，可以参考 3.2.1 节所介绍的方法，在给定的命令行上，添加 `-XX:+Printflagsfinal`（默认为 `false`，即“关闭”）就能获得具体运行环境中特定标志的默认值。基于环境对标志进行自动调优的过程称为*自动优化*（Ergonomics）。

> **Client 和 Server 类虚拟机**
>
> Java 的自动优化前提是机器被分为“Client”和“Server”。这两个术语直接与特定平台（参见第 4 章）上的默认 JVM 编译器相关，它们也设定了默认的调优标志。例如，机器类别决定了平台默认的垃圾收集器（参见第 5 章）。
>
> Microsoft Windows 上运行的任何 32 位 JVM（无论机器上 CPU 的个数是多少），以及单 CPU 机器（不论是什么操作系统）上运行的任何 32 位 JVM，都是 Client 类机器。所有其他机器（包括所有 64 位 JVM）都被认为是 Server 类。

从 Oracle 和 OpenJDK 站点下载的 JVM，被称为“产品”JVM。需要生成不同版本的 JVM 时，可以从源代码构建：调试版、开发版等。这些版本通常有更多的特性。特别是开发版，包含了大量的调优标志，开发人员可以尝试与 JVM 各种算法的细节打交道。不过本书基本上不会考虑这些标志。

1.3 全面的性能调优

本书关注于如何以最佳方式利用 JVM 和 Java 平台 API，让程序运行得更快。但除了这两点，还有许多外在的因素影响性能。书中这些因素时不时会出现，但因为它们不只影响 Java，所以不会深入讨论。JVM 和 Java 平台的性能只是高性能主题中的一小部分。

本书会覆盖一些外部因素，这些因素的重要性不亚于 Java 的性能调优。本书中基于 Java 的调优方法可以和这些因素相互补充，但这些因素多数已经超过了本书讨论的范围。

1.3.1 编写更好的算法

Java 的许多细节和性能标志都可以影响应用的性能，只不过从来都没有一个叫 `-XX:+RunReallyFast` 的神奇标志。

归根结底，应用的性能取决于它的代码如何编写。例如，如果程序循环遍历数组中的所有元素，JVM 就可以优化数组的边界检查，使循环更快，展开循环能提供额外的加速。但如果循环是为了找到特定元素，那目前还没有什么优化的办法，使得遍历数组和采用 `HashMap` 的版本一样快。

需要更高性能时，算法是否优秀就是重中之重了。

1.3.2 编写更少的代码

有些人写代码是为钱，有些是为乐趣，还有些人将代码回馈社区，但不管怎样，大家都是码农（或者在写程序的团队里工作）。很难想象，我们对项目的贡献是少写代码，因为仍然有管理者通过所写的代码量来评估开发人员的绩效。

我能理解这种想法，不过这种想法与现实并不吻合。同样是正确的程序，小程序运行起来要比大程序快。对所有的计算机程序来说都是如此，Java 程序自然也不例外。要编译的代码越多，等待程序启动所耗费的时间就越长；要创建和销毁的对象越多，垃圾收集的工作

量就越大；要分配和持有的对象越多，GC 的周期就越长；要从磁盘装载进 JVM 的类越多，程序启动所花费的时间就越长；要执行的代码越多，机器硬件缓存的效率就越低；而执行的代码越多，花费的时间就越长。

> **无法取胜的战争**
>
> 与直觉相反（和令人沮丧）的是，所有应用的性能都会随着时间，即应用新版本的发布而降低。但由于硬件的改善使得新程序的运行速度可以被接受，所以通常都不会有人注意到性能上的差异。
>
> 想象一下，在曾经运行 Windows 95 的机器上运行 Windows Aero 界面，会是什么样子？我以前喜欢 Mac Quadra 950，但它无法运行 Mac OS X（如果真这么做了，它将比 Mac OS 7.5 慢许多许多）。从更小的层次上看，Firefox 23.0 比 Firefox 22.0 快，但它们之间的版本差别很小。具有按 tab 页浏览、同步滚动和安全特性的 Firefox 要比之前的 Mosaic 强大，但 Mosaic 从我硬盘里装载基本 HTML 文件的速度比 Firefox 23.0 快 50%。
>
> 当然，Mosaic 几乎不能从任何的热门网站上装载实际的 URL，所以不太可能把 Mosaic 作为主要的浏览器。一般来说，特别是在两个小版本之间，代码会进行优化，从而运行得更快。性能优化工程师应该注意到这点。如果我们擅长这份工作，那就能赢得这场战斗。这是美好而有意义的事。我认为我们应该改善现有应用的性能。
>
> 但铁一般的事实是：随着新特性的添加和新要求的采纳（为了与对手竞争），程序会越来越大，越来越慢。

我把这总结为“积少成多”原则。开发人员总争辩说，只是增加了很小的功能，压根就不会有什么时间损耗（特别是不使用该功能的时候）。接着项目中的其他开发人员也同样拍着胸脯保证，结果却发现性能突然下降了好几个百分点。下次发布的时候又重复出现这样的情景，而此时程序性能已经下降了 10%，反复几次这样的过程之后，性能测试就会检测到资源瓶颈——内存使用达到临界点、代码缓存溢出等情况。对于这些情形，常规的性能测试可以捕获发生状况的原因，性能调优小组也可以修正主要的性能衰减。但随着时间的推移，小衰减积少成多，会越来越难以修复。

我并不是在鼓吹永远不要为产品增加新特性或者新代码，很显然增强程序是有利可图的。但你得小心权衡，尽可能提高效能。

1.3.3　老调重弹的过早优化

“过早优化”一词公认是由高德纳发明的，开发人员常常据此宣称：只有在运行时才能知道代码的性能有多要紧。但你可能从来没注意到，完整的原话是“我们不应该把大量时间都耗费在那些小的性能改进上；过早考虑优化是所有噩梦的根源”。

这句名言的重点是，最终你应该编写清晰、直接、易读和易理解的代码。这里的“优化”应该理解为虽然算法和设计改变了复杂程序的结构，但是提供了更好的性能。那些真正的优化最好留到以后，等到性能分析表明这些措施有巨大收益的时候才进行。

而这里所指的过早优化，并不包括避免那些已经知道对性能不好的代码结构。每行代码，如果有两种简单、直接的编程方式，那就应该选择性能更好的那种。

在某种程度上，有经验的 Java 开发人员都能很好地领会到这点（这也是一个例证，说明他们日积月累而掌握了调优艺术）。思考以下代码：

```
log.log(Level.FINE, "I am here, and the value of X is"
        + calcX() + " and Y is " + calcY());
```

代码包含了一个看起来不太必要的字符串连接。因为除非日志级别很高，否则字符串的信息并不会记录到日志中，如果不打印日志消息，那就没必要调用 `calcX()` 和 `calcY()`。有经验的 Java 开发人员会下意识地避免这种写法。有些 IDE（例如 NetBeans）会在代码上打标记并建议更改。（然而没有完美的工具：NetBeans 会在字符串连接操作上打标记，却不会建议去掉不必要的方法调用。）

像这样的日志代码会更好：

```
if (log.isLoggable(Level.FINE)) {
    log.log(Level.FINE,
            "I am here, and the value of X is {} and Y is {}",
            new Object[]{calcX(), calcY()});
}
```

除非启用了日志功能，否则就可以在避免字符串连接（消息体中有格式化字符，不会提高性能，但使代码更清晰）的同时，避免方法调用或者对象分配。

这样写出来的代码仍然清晰易读，与原来的代码相比，没有太多额外工作。好吧，我们还是需要多敲几下键盘，多加一行逻辑。不过这仍然不属于应该避免的过早优化，它是好码农所熟悉的选择。在你思考如何写代码的时候，请不要生搬硬套前辈们的教条。

本书中我们还会看到其他例子，例如第 9 章讨论了处理 `Vector` 前先进行循环的性能。

1.3.4　其他：数据库很可能就是瓶颈

如果你开发的是独立运行不使用外部资源的 Java 应用，性能就（几乎）只与应用本身相关。一旦添加了外部资源（例如数据库），那这两者的性能就都很重要了。在分布式环境中，比如 Java EE 应用服务器、负载均衡器、数据库和后台企业信息系统，Java 应用服务器的性能问题可能只是其中很小的部分。

本书并不关注整体系统的性能。对于整体系统，我们需要采取结构化方法针对系统的所有方面分析性能。CPU 使用率、I/O 延迟、系统整体的吞吐量都必须测量和分析。只有到那时，我们才能判定到底是哪个组件导致了性能瓶颈。关于这个主题有大量优秀的资源，相关的方法和工具也不只针对 Java。假定你已经完成了分析，并且判断出是运行环境中 Java 组件的性能需要改善。

> **不只 JVM 有 bug 和性能问题**
>
> 这节以数据库的性能为例，但运行环境的任何部分都可能会引起性能问题。
>
> 我曾经遇到过一个问题，客户正在安装新版本的应用服务器，而测试显示请求发送到服务器上的时间变得越来越长。于是我根据奥卡姆剃刀原则（参见下一条贴士），考察应用服务器中所有可能产生问题的部分。
>
> 逐一排除之后，性能问题依旧，而且我也没发现后台数据库有问题。因此最可能的原因是测试框架，通过性能分析判定负载发生器——Apache JMeter——才是性能衰退的原因。它将每个响应保留在列表中，每次有新响应到来时，它都要遍历整个列表，以便找到响应时间 90% 的请求（如果不熟悉这些词，请参见第 2 章）。
>
> 部署应用的系统，它的任何部分都可能会引起性能问题。常规案例分析建议应该首先考虑系统最新变动的部分（通常是 JVM 中的应用），但仍然要准备检查环境的每一个可能出现问题的组件。

另一方面，不要忽视初步分析。如果数据库是瓶颈（提示：的确是的话），那么无论怎么优化访问数据库的 Java 应用，都无助于整体性能；实际上可能适得其反。作为一般性原则，系统负载增加越大，系统性能就会越糟糕。如果更改了 Java 应用使得它更有效，这只会增加已经过载的数据库的负载，整体性能实际反而会下降。导致的风险是，可能会得出错误结论，即认为不应该改进 JVM。

增加系统某个组件的负载从而导致整个系统性能变慢，这项原则不仅限于数据库。CPU 密集型的应用服务器增加负载，或者越来越多线程试图获取已经有线程等待的锁，还有许多其他场景，也都适用这项原则。第 9 章展示了一个仅涉及 JVM 的极端例子。

1.3.5 常见的优化

如果所有的性能问题同等重要，从而“积少成多”地改进性能，那是多么吸引人。但常见的用例场景才是真正应该关注的重点。

我们可以从以下几方面阐述这条原则。

- 借助性能分析来优化代码，重点关注性能分析中最耗时的操作。然而请注意，这并不意味着只看性能分析中的叶子方法（参见第 3 章）。
- 利用奥卡姆剃刀原则诊断性能问题。性能问题最可能的原因应该是最容易解释的：新代码比机器配置更可能引入性能问题，而机器配置比 JVM 或者操作系统的 bug 更容易引入性能问题。隐藏的 bug 确实存在，但不应该把最可能引起性能问题的原因首先归咎于它，而只在测试用例通过某种方式触发了隐藏的 bug 时才关注。但不应该一上来就跳到这种不太可能的场景。
- 为应用中最常用的操作编写简单算法。以估算数学公式的程序为例，用户可以决定他所期望的最大容许误差为 10% 或 1%。如果 10% 的误差适合多数用户，那么优化代码就意味着即便误差范围缩小为 1%，但是速度变慢了。

1.4 小结

Java 7 和 Java 8 引入了大量新特性和工具，使得 Java 应用的性能更容易发挥到极致。本书有助于你理解如何有效地利用所有的 JVM 特性，最终使程序如虎添翼。

不过请记住，许多情况下，JVM 只占整体性能的一小部分。你需要对 Java 所在的环境进行整体系统调优，数据库和其他后台运行系统性能的重要性不亚于 JVM。不过整体的性能分析不是本书的关注重点，本书假设我们已经做过详细的调查，确定环境中的 Java 组件是系统的重要瓶颈。

此外，JVM 与系统其他部分的交互对性能的影响也同样重要，无论是直接交互（例如以最佳方式使用 JDBC），还是间接交互（例如优化应用所使用的本地内存，这类应用与大型系统的各种组件共享机器）。本书也有助于解决这类性能问题。

第 2 章

性能测试方法

本章讨论了性能测试的 4 项原则。这些原则是后续章节的基础，也涵盖了性能工程科学的各个方面。

后续章节中的许多示例均取材于一个普通应用，本章也对此做了概要介绍。

2.1　原则1：测试真实应用

第 1 条原则就是，应该在产品实际使用的环境中进行性能测试。性能测试大体上可以分为 3 种，每种都有其优点和不足，只有适用于实际应用的才能取得最好的效果。

2.1.1　微基准测试

第 1 种是微基准测试。微基准测试用来测量微小代码单元的性能，包括调用同步方法的用时与非同步方法的用时比较，创建线程的代价与使用线程池的代价，执行某种算法的耗时与其替代实现的耗时，等等。

微基准测试看起来很好，但要写对却很困难。考虑以下代码，被测的方法是计算出第 50 个斐波那契数，这段代码试图用微基准测试来测试不同实现的性能：

```
public void doTest() {
    // 主循环
    double l;
    long then = System.currentTimeMillis();
    for (int i = 0; i < nLoops; i++) {
        l = fibImpl1(50);
    }
    long now = System.currentTimeMillis();
    System.out.println("Elapsed time: " + (now - then));
```

```
    }
    ...
    private double fibImpl1(int n) {
        if (n < 0) throw new IllegalArgumentException("Must be > 0");
        if (n == 0) return 0d;
        if (n == 1) return 1d;
        double d = fibImpl1(n - 2) + fibImpl(n  - 1);
        if (Double.isInfinite(d)) throw new ArithmeticException("Overflow");
        return d;
    }
```

代码看起来简单，却存在很多问题。

1. 必须使用被测的结果

这段代码的最大问题是，实际上它永远都不会改变程序的任何状态。因为斐波那契的计算结果从来没有被使用，所以编译器可以很放心地去除计算结果。智能的编译器（包括当前的 Java 7 和 Java 8）最终执行的是以下代码：

```
long then = System.currentTimeMillis();
long now = System.currentTimeMillis();
System.out.println("Elapsed time: " + (now - then));
```

结果是，无论计算斐波那契的方法如何实现，循环执行了多少次，实际的流逝时间其实只有几毫秒。循环如何被消除的细节请参见第 4 章。

有个方法可以解决这个问题，即确保读取被测结果，而不只是简单地写。实际上，将局部变量 `l` 的定义改为实例变量（并用关键字 `volatile` 声明）就能测试这个方法的性能了。（实例变量 `l` 必需声明为 `volatile` 的原因请参见第 9 章。）

> **多线程微基准测试**
>
> 即便本示例是单线程微基准测试，也必需使用 `volatile` 变量。
>
> 编写多线程微基准测试时务必深思熟虑。当若干个线程同时执行小段代码时，极有可能会产生同步瓶颈（以及其他线程问题）。所以，如果我们过多依赖多线程基准测试的结果，就常常会将大量时间花费在优化那些真实场景中很少出现的同步瓶颈上，而不是性能需求更迫切的地方。
>
> 考虑这样的微基准测试，即有两个线程同时调用同步方法。由于基准测试的代码量相对于被测方法来说比较少，所以多数时间都是在执行同步方法。假设执行同步方法的时间只占整个微基准测试的 50%，即便少到只有两个线程，同时执行同步代码的概率仍然很高。因此基准测试运行得很慢，并且随着线程数的增加，竞争所导致的性能问题将愈演愈烈。最终结果就是，测试衡量的是 JVM 如何处理竞争，而不是微基准测试的本来目的。

2. 不要包括无关的操作

即便使用了被测结果，依然还有隐患。上述代码只有一个操作：计算第 50 个斐波那契数。可想而知，其中有些迭代操作是多余的。如果编译器足够智能的话，就能发现这个问题，

从而只执行一遍循环——至少可以少几次迭代，因为那些迭代是多余的。

另外，`fibImpl(1000)` 的性能可能与 `fibImpl(1)` 相差很大。如果目的是为了比较不同实现的性能，测试的输入就应该考虑用一系列数据。

也就是说，解决这个问题，需要给 `fibImpl1()` 传入不同的参数。可以使用随机值，但仍然必须小心。

下面是种简单方法，即在循环中使用随机数生成器：

```
for (int i = 0; i < nLoops; i++) {
    l = fibImpl1(random.nextInteger());
}
```

可以看到，循环中包括了计算随机数，所以测试的总时间是计算斐波那契数列的时间，加上生成一组随机数的时间。这可不是我们的目的。

微基准测试中的输入值必须事先计算好，比如：

```
int[] input = new int[nLoops];
for (int i = 0; i < nLoops; i++) {
    input[i] = random.nextInt();
}
long then = System.currentTimeMillis();
for (int i = 0; i < nLoops; i++) {
    try {
        l = fibImpl1(input[i]);
    } catch (IllegalArgumentException iae) {
    }
}
long now = System.currentTimeMillis();
```

3. 必须输入合理的参数

此处还有第 3 个隐患，就是测试的输入值范围：任意选择的随机输入值对于这段被测代码的用法来说并不具有代表性。在这个测试例子中，有一半的方法调用会立即抛出异常（即所有的负数）。输入参数大于 1476 时，也都会抛出异常，因为此时计算出来的是 double 类型所能表示的最大的斐波那契数。

如果计算斐波那契数的速度大幅度提升，但例外情况直到计算结束时才被监测到时，在实现中会发生什么？考虑下面这种替代实现：

```
public double fibImplSlow(int n) {
    if (n < 0) throw new IllegalArgumentException("Must be > 0");
    if (n > 1476) throw new ArithmeticException("Must be < 1476");
    return verySlowImpl(n);
}
```

虽然很难想象会有比原先用递归更慢的实现，但我们不妨假定有这么个实现并用在了这段代码里。通过大量输入值比较这两种实现，我们会发现，新的实现竟然比原先的实现快得多——仅仅是因为在方法开始时进行了范围检查。

如果在真实场景中，用户只会传入小于 100 的值，那这个比较就是不正确的。通常情况下 `fibImpl()` 会更快，正如第 1 章所说，我们应该为常见的场景进行优化。（显然这是个精心

构造的例子。不管怎样，仅仅在原先的实现上添加了边界测试就使得性能变好，通常这是不可能的。）

> **热身期**
>
> Java 的一个特点就是代码执行的越多性能越好，第 4 章将会覆盖这个主题。基于这点，微基准测试应该包括热身期，使得编译器能生成优化的代码。
>
> 本章后续将深入讨论热身期的优缺点。微基准测试需要热身期，否则测量的是编译而不是被测代码的性能了。

综合所有因素，正确的微基准测试代码看起来应该是这样：

```
package net.sdo;

import java.util.Random;

public class FibonacciTest {
    private volatile double l;
    private int nLoops;
    private int[] input;

    public static void main(String[] args) {
        FibonacciTest ft = new FibonacciTest(Integer.parseInt(args[0]));
        ft.doTest(true);
        ft.doTest(false);
    }

    private FibonacciTest(int n) {
        nLoops = n;
        input = new int[nLoops];
        Random r = new Random();
        for (int i = 0; i < nLoops; i++) {
            input[i] = r.nextInt(100);
        }
    }

    private void doTest(boolean isWarmup) {
        long then = System.currentTimeMillis();
        for (int i = 0; i < nLoops; i++) {
            l = fibImpl1(input[i]);
        }
        if (!isWarmup) {
            long now = System.currentTimeMillis();
            System.out.println("Elapsed time:" + (now - then));
        }
    }

    private double fibImpl1(int n) {
        if (n < 0) throw new IllegalArgumentException("Must be > 0");
        if (n == 0) return 0d;
        if (n == 1) return 1d;
        double d = fibImpl1(n - 2) + fibImpl(n  - 1);
```

```
            if (Double.isInfinite(d)) throw new ArithmeticException("Overflow");
            return d;
        }
    }
```

甚至这个微基准测试的测量结果中也仍然有一些与计算斐波那契数没有太大关系：调用 fibImpl1() 的循环和方法开销，将每个结果都写到 volatile 变量中也会有额外开销。

此外还需要留意编译效应。编译器编译方法时，会依据代码的性能分析反馈来决定所使用的最佳优化策略。性能分析反馈基于以下因素：频繁调用的方法、调用时的栈深度、方法参数的实际类型（包括子类）等，它还依赖于代码实际运行的环境。编译器对于相同代码的优化在微基准测试中和实际应用中经常有所不同。如果用相同的测试衡量斐波那契方法的其他实现，就能看到各种编译效应，特别是当这个实现与当前的实现处在不同的类中时。

最终，还要探讨微基准测试实际意味着什么。比如这里讨论的基准测试，它有大量的循环，整体时间以秒计，但每轮循环迭代通常是纳秒级。没错，纳秒累计起来，“积少成多”就会成为频繁出现的性能问题。特别是在做回归测试的时候，追踪级别设为纳秒很有意义。如果集合操作每次都节约几纳秒，日积月累下来意义就很重大了（示例参见第 12 章）。对于那些不频繁的操作来说，例如那种同时只需处理一个请求的 servlet，修复微基准测试所发现的纳秒级性能衰减就是浪费时间，这些时间用在优化其他操作上可能会更有价值。

微基准测试难于编写，真正管用的又很有限。所以，应该了解这些相关的隐患后再做出决定，是微基准测试合情合理值得做，还是关注宏观的测试更好。

2.1.2　宏基准测试

衡量应用性能最好的事物就是应用自身，以及它所用到的外部资源。如果正常情况下应用需要调用 LDAP 来检验用户凭证，那应用就应该在这种模式下测试。虽然删空 LDAP 调用在模块测试中有一定意义，但应用本身必须在完整真实配置的环境中测试。

随着应用规模的增长，上述准则愈加重要也更难达到。复杂系统并不是各个部分的简单加和，装配之后，各部分的行为会有很大不同。所以，比如你伪装数据库调用，那就意味着你并不担心数据库的性能——对了，你是 Java 人，为什么要处理其他人的性能问题呢？数据库连接会因为缓存而消耗大量堆内存，网络也会因为发送大量数据而饱和，代码调用简单方法（与调用 JDBC 驱动程序的代码相比）时的不同优化，短代码路径因为 CPU 管线和缓存而比长代码路径更为有效，等等。

需要测试整体应用的另外一个原因是资源的分配。在完美世界中，我们有足够的时间去优化应用的每一行代码。但现实是，截止日期迫在眉睫，只对复杂系统进行部分优化也无法立即奏效。

考虑图 2-1 中的数据流。用户发起数据请求，然后系统进行业务处理，并基于结果从数据库装载数据，再进行处理，最后将更改后的数据存入数据库，并将结果发还给用户。方框中的数字（例如 200 RPS）是每秒的请求数，是模块单独测试时所能承载的处理量。

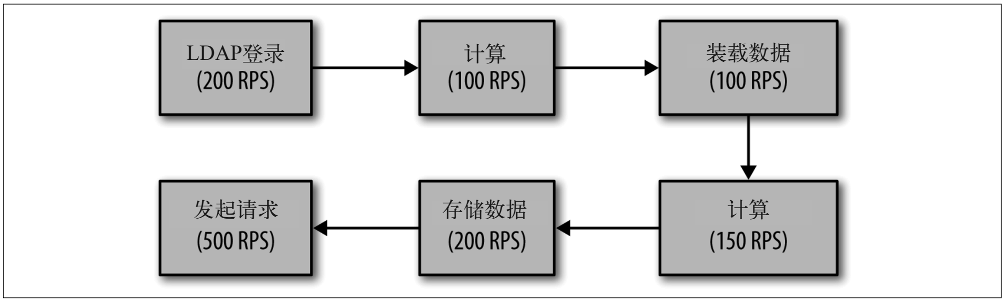


图 2-1：典型的程序流程

从商业角度看，业务处理是最重要的，是程序存在的理由，也是有人愿意付钱给我们的原因。不过在这个例子中，即便业务处理速度提高 100% 也完全没什么好处。任何应用（包含独立运行的 JVM）都可以像这样划分成一系列步骤，方框中的模块、子系统等产生数据的速度取决于它们的效率。（在这个模型中，每个方框耗费的时间包括子系统代码的执行时间，也包括网络传输的时间、磁盘传输的时间，等等。如果是模块化的模型，时间应该只包括该模块内代码的执行时间。）数据进入子系统的速率取决于前一个模块或系统的输出速率[1]。

假设业务处理的算法有所改进，处理量达到了 200 RPS，系统能承受的负载也相应增加。LDAP 系统可以处理这些增加的负载：目前为止一切都好，数据将以 200 RPS 的速率注入业务处理模块，而它也将以 200 RPS 的速率输出。

但数据库只能以 100 RPS 的速率装载数据。虽然向数据库发送请求的速率为 200 RPS，输出到其他模块的速率却只有 100 RPS。即便业务逻辑处理的效率加倍，系统整体的吞吐量仍然只能达到 100 RPS。所以，除非花时间改善环境其他方面的效率，否则业务逻辑做再多改进也是无效的。

多 JVM 时的全系统测试

全应用测试有个很重要的场景，就是同一台机器上同时运行多个应用。许多 JVM 的默认调优都是默认假定整个机器的资源都归 JVM 使用。如果单独测试，优化效果很好。如果在其他应用（包括但不限于 Java 程序[2]）运行的时候进行测试，性能会有很大的不同。

这方面的示例请参见后续章节，这里只快速过一遍：单个 JVM（默认配置）执行 GC 周期时，该机器上所有处理器的 CPU 使用率都会变成 100%。如果测量程序执行时的平均 CPU 使用率，大概会有 40%——实际意思是，某些时候 30% 的 CPU 被占用，其他时候为 100%。当隔离 JVM 时，它可以运行得很好，但如果 JVM 与其他应用并发运行，它就不可能在 GC 时获得 100% 的 CPU。此时测出来的性能会与它单独运行时不同。

这是微基准测试和模块测试不可能让你全面了解应用性能的另一个原因。

注 1：原文中的“box”指图中的方框，其含义是模块或子系统，为便于理解，此处采取意译。——译者注
注 2：原文“other JVM”直译容易误解为“其他 JVM 实现”，此处改用“Java 程序”。——译者注

本例子中，优化业务处理并不完全是浪费时间：在系统其他性能瓶颈上曾经付出的努力，终究会有好处。进一步说，这中间有个优先顺序：不进行整体应用的测试，就不可能知道哪部分的优化会产生回报。

2.1.3 介基准测试

我的调优工作包括 Java SE 和 EE，每种都会有一组类似微基准测试的测试。对于 Java SE 工程师来说，这个术语意思是样本甚至比 2.1.1 节的还要小：测量很小的东西。Java EE 工程师则将这个术语用于其他地方：测量某方面性能的基准测试，但仍然要执行大量代码。

Java EE 微基准测试的例子，测量从应用服务器返回的简单 JSP 响应。仔细比较处理请求的代码和传统微基准测试的代码：有许多 socket 管理代码，读取请求、查找（可能需要编译）JSP、写入响应等代码。从传统角度来看，这不能算微基准测试。

这种测试也不是宏基准测试：没有安全（比如用户不用登录），没有会话管理，也没有大量使用其他的 Java EE 特性。因为它只是实际应用的子集，介于两者之间——它是介基准测试。介基准测试并不局限于 Java EE：它是一个术语，我用来表示做一些实际工作，但不是完整应用的基准测试。

介基准测试与微基准测试相比隐患更少，又比宏基准测试容易。介基准测试不包含会被编译器优化的大量死代码（除非应用中真的存在死代码，否则这种情况下优化是件好事）。介基准测试更容易线程化：它们比全应用时运行的代码更容易遇到同步瓶颈，不过这些是实际应用在更大规模硬件系统和更大负载时，最终都会遇到的瓶颈。

介基准测试仍然不完美。开发人员用这样的基准测试比较两个应用服务器性能时，容易误入歧途。考虑表 2-1 中两个应用服务器假想的响应时间。

表2-1：两个应用服务器的假想响应时间

测试	应用服务器1（毫秒）	应用服务器2（毫秒）
简单 JSP	19	50
有会话的 JSP	75	50

仅使用简单 JSP 的开发人员比较两个服务器性能时，可能不会意识到，服务器 2 会自动为每个请求创建会话。他可能会得出服务器 1 性能更快的结论，结果他就做出了错误选择，因为实际上服务器 1 创建会话要花费更长时间。（后续调用的性能是否有差别是另一个需要考虑的因素，但从这些数据无法预计一旦会话创建后，哪个服务器的性能会更好。）

即便如此，介基准测试也为测试全应用提供了一个合理选择。它们的性能比微基准测试更接近实际应用。这里有个连续的过程。本章稍后的章节将概要介绍一个常见应用，后续章节中的许多示例程序都出自该应用。这个应用有 EE 模式，但这种模式不使用会话复制（高可用），或者基于 EE 平台的安全。虽然它能访问企业资源（比如数据库），但多数示例中它只使用随机数据。在 SE 模式下，它模仿一些实际（但很快）计算：比如没有 GUI 或者用户交互发生。

介基准测试也有益于自动化测试，特别是模块级别的测试。

快速小结

1. 好的微基准测试既难写，价值又有限。如果你必须使用它，那可以用它来快速了解性能，但不要依赖它们。
2. 测试完整应用是了解它实际运行的唯一途径。
3. 在模块或者操作级别隔离性能——介基准测试——相对于全应用测试来说，是一种合理的折中途径，而不是替代方法。

2.1.4 代码示例

贯穿全书的许多例子都来自于一个示例应用，计算某只股票在一段时间内的“历史”最高价和最低价，以及标准差。因为所有数据皆为虚构，价格和股票代码也是随机生成，所以这里的历史标上了引号。

本书的所有示例代码都可在我的 GitHub 上[3]找到，本节只是覆盖了代码的基本要点。基本接口 `StockPrice` 表示某股票某天的价格区间：

```
public interface StockPrice {
        String getSymbol();
        Date getDate();
        BigDecimal getClosingPrice();
        BigDecimal getHigh();
        BigDecimal getLow();
        BigDecimal getOpeningPrice();
        boolean isYearHigh();
        boolean isYearLow();
        Collection<? extends StockOptionPrice> getOptions();
}
```

通常，那些示例应用都是对一组股价进行处理，这些股价表示一段时间内的股票历史（比如 1 年或 25 年，取决于具体的示例）：

```
public interface StockPriceHistory {
        StockPrice getPrice(Date d);
        Collection<StockPrice> getPrices(Date startDate, Date endDate);
        Map<Date, StockPrice> getAllEntries();
        Map<BigDecimal,ArrayList<Date>> getHistogram();
        BigDecimal getAveragePrice();
        Date getFirstDate();
        BigDecimal getHighPrice();
        Date getLastDate();
        BigDecimal getLowPrice();
        BigDecimal getStdDev();
        String getSymbol();
}
```

这个接口的基本实现是从数据库载入股价：

```
public class StockPriceHistoryImpl implements StockPriceHistory {
    ...
```

注 3：https://github.com/ScottOaks/JavaPerformanceTuning。——译者注

```
    public StockPriceHistoryImpl(String s, Date startDate,
        Date endDate, EntityManager em) {
        Date curDate = new Date(startDate.getTime());
        symbol = s;
        while (!curDate.after(endDate)) {
            StockPriceImpl sp = em.find(StockPriceImpl.class,
                        new StockPricePK(s, (Date) curDate.clone()));
            if (sp != null) {
                Date d = (Date) curDate.clone();
                if (firstDate == null) {
                    firstDate = d;
                }
                prices.put(d, sp);
                lastDate = d;
            }
            curDate.setTime(curDate.getTime() + msPerDay);
        }
    }
    ...
}
```

这个示例的架构是从数据库载入数据，第 11 章将使用这个功能。不过为了便于运行示例，多数时候将用伪装过的实体管理器（mock entity manager）随机生成一系列数据。大体上，多数示例是模块级别的介基准测试，适合随手演示性能问题——不过全应用运行时，我们会了解实际的应用性能（参见第 11 章）。

附带说明一下，许多示例依赖随机数生成器的性能。与微基准测试示例不同，这里是有意为之，可以展示一些 Java 的性能问题。（就此而言，示例是为了测量一些任意状况下的性能，随机数生成器的性能正好适合此目的。这点与微基准测试大有不同，微基准测试中包括随机数生成时间就会影响整体计算。）

这些示例重度依赖 `BigDecimal` 的性能，它被用来存储所有的数据。这是保存货币数据时的标准选择，如果货币用原生的 `double` 对象，半分钱的舍入和更小的数量就会很成问题。从编写示例的角度来看，这样做也有价值，因为可以在计算股价标准差时产生一些“业务逻辑”或者延长计算。计算标准差需要知晓 `BigDecimal` 数的平方根。标准 Java API 不支持这个函数，示例将采用以下方法：

```
public static BigDecimal sqrtB(BigDecimal bd) {
    BigDecimal initial = bd;
    BigDecimal diff;
    do {
        BigDecimal sDivX = bd.divide(initial, 8, RoundingMode.FLOOR);
        BigDecimal sum = sDivX.add(initial);
        BigDecimal div = sum.divide(TWO, 8, RoundingMode.FLOOR);
        diff = div.subtract(initial).abs();
        diff.setScale(8, RoundingMode.FLOOR);
        initial = div;
    } while (diff.compareTo(error) > 0);
    return initial;
}
```

这是巴比伦平方根计算法的实现。它不是最有效的实现，特别是初始值可以估算得更好，

可以少几轮迭代。这是经过深思熟虑的，因为计算需要花费一些时间（模拟业务逻辑），不过它展示了第 1 章中的基本观点：使 Java 代码更快的常用方法是更好的算法，这不依赖 Java 调优或者 Java 编码实践。

StockPriceHistory 接口实现中的标准差、平均值和直方图，都是由具体数据推演出来的。在不同的实现中，会立即计算（从实体管理器加载数据的时候）或者推迟计算（调用该方法的时候）。StockPrice 所引用的 StockOptionPrice 与此类似，它是股票在特定天的期权价之一，它的值也可以立即或者推迟从实体管理器中获取。对于这两种场景，通过接口定义，不同的实现可以在不同情况下进行比较。

这些接口也与 Java EE 应用自然吻合：用户先访问 JSP 页面，然后选择感兴趣的股票代码和时间范围。在标准示例中，请求将发送到标准 servlet，它会解析输入参数，通过内嵌的 Java Persistence API（JPA）调用无状态的 Enterprise JavaBean（EJB），以获取数据，然后将响应转发到 JavaServer Pages（JSP）页面，它再将数据格式化成 HTML 的形式：

```
protected void processRequest(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
    try {
        String symbol = request.getParameter("symbol");
        if (symbol == null) {
        symbol = StockPriceUtils.getRandomSymbol();
        }
            ... similar processing for date and other params...
        StockPriceHistory sph;
        DateFormat df = localDf.get();
        sph = stockSessionBean.getHistory(symbol, df.parse(startDate),
                        df.parse(endDate), doMock, impl);
    String saveSession = request.getParameter("save");
    if (saveSession != null) {
        .... Store the data in the user`s session ....
        .... Optionally store the data in a global cache for
        .... use by other requests
    }
    if (request.getParameter("long") == null) {
        // 发回一个带有约4K大小的数据的页面
        request.getRequestDispatcher("history.jsp").
                forward(request, response);
    }
    else {
        // 发回一个带有约100K大小的数据的页面
        request.getRequestDispatcher("longhistory.jsp").
                forward(request, response);
        }
    }
```

这个类可以注入不同的 StockPriceHistory 实现（除了其他方法，初始化方法立即执行或者推迟）。它可以选择缓存后端数据库（或者是伪装的实体管理器）的数据。处理企业应用时，这些都是通常的做法（特别是，中间层应用服务器可以缓存数据，有时被认为是它很大的性能优势）。贯穿全书的例子也将解释这些权衡。

被测系统的硬件

虽然本书主要集中在软件上，但基准测试同样也会测量它们所运行的硬件。

本书中多数的例子都运行在我的台式机系统上，CPU 为 4 核（4 个逻辑 CPU）的 AMD Athlon X4 640 CPU，物理内存为 8 GB，操作系统为 Ubuntu Linux 12.04 LTS。

2.2 原则2：理解批处理流逝时间、吞吐量和响应时间

性能测试的第 2 条原则是多角度审视应用性能。应该测量哪个指标取决于对应用最重要的因素。

2.2.1 批处理流逝时间

测量应用性能的最简单方法是，看它完成任务花了多少时间，例如接收 10 000 只股票 25 年的历史价格并计算标准差，生成某公司 50 000 名雇员的薪酬福利报表，以及执行 1 000 000 次循环的时间等。

在非 Java 的世界，可以很直接地测试流逝时间：应用记下时间点从而测量执行时间。但在 Java 世界中，由于即时编译（JIT），这种方法就会有些问题了。第 4 章描述了这个过程，其中的重点是，虚拟机会花几分钟（或更长时间）全面优化代码并以最高性能执行。由于这个（以及其他）原因，研究 Java 的性能优化就要密切注意代码优化的热身期：大多数时候，应该在运行代码执行足够长时间，已经编译并优化之后再测量性能。

其他影响应用热身的因素

通常认为的应用热身，指的就是等待编译器优化运行代码，不过基于代码的运行时长还有其他一些影响性能的因素。

例如，JPA 通常都会缓存从数据库读取的数据（参见第 11 章），由于这些数据可以从缓存中获取而不需要长途跋涉到数据库，所以通常再次使用时，操作就会变得更快。与此类似，应用程序读文件时，操作系统就会将文件载入内存。随后再次读取相同文件就会变得更快，这是因为数据已经驻留在计算机主内存中，并不需要从磁盘实际读取。

一般来说，应用热身过程中有许多地方会缓存数据，虽然并不都那么明显。

另一方面，许多情况下应用从开始到结束的整体性能更为重要。报告生成器处理 10 000 个数据元素需要花费大量时间，但对最终用户而言，处理前 5000 个元素是否比后 5000 个慢 50% 并不重要。即便是像应用服务器这样的系统——其性能必定会随运行时间而改善——初始的性能依然很重要。某种配置下的应用服务器需要 45 分钟才能达到性能峰值。对于

在这段时间访问应用的用户来说，热身期的性能就很重要了。

基于这些理由，本书的许多例子都是面向批处理的（即便这看起来有些不寻常）。

2.2.2 吞吐量测试

吞吐量测试是基于一段时间内所能完成的工作量。虽然最常见的吞吐量测试是服务器处理客户端产生的数据，但这并非绝对的：单个独立运行的应用也可以像测量流逝时间一样测量吞吐量。

在客户端 – 服务器的吞吐量测试中，并不考虑客户端的思考时间。客户端向服务器发送请求，当它收到响应时，立刻发送新的请求。持续这样的过程，等到测试结束时，客户端会报告它所完成的操作总量。客户端常常有多个线程在处理，所以吞吐量就是所有客户端所完成的操作总量。通常这个数字就是每秒完成的操作量，而不是测量期间的总操作量。这个指标常常被称作每秒事务数（TPS）、每秒请求数（RPS）或每秒操作次数（OPS）。

所有的客户端 - 服务器测试都存在风险，即客户端不能足够快地向服务器发送数据。这可能是由于客户端机器的 CPU 不足以支持所需数量的客户端线程，也可能是因为客户端需要花大量时间处理响应才能发送新的请求。在这些场景中，测试衡量的其实是客户端性能而不是服务器性能，这并不是我们的目的。

其中的风险依赖于每个线程所承载的工作量（客户端机器的线程数和配置）。由于客户端线程需要执行大量工作，零思考时间（面向吞吐量）测试更可能会遇到这种情形。因此，通常吞吐量测试比响应时间测试的线程数少，线程负载也小。

通常吞吐量测试也会报告请求的平均响应时间。这是重要的信息，但它的变化并不表示性能有问题，除非报告的吞吐量相同。能够承受 500 OPS、响应时间 0.5 秒的服务器，它的性能要好过响应时间 0.3 秒但只有 400 OPS 的服务器。

吞吐量测试总是在合适的热身期之后进行，特别是因为所测量的东西并不固定。

2.2.3 响应时间测试

最后一个常用的测试指标是响应时间：从客户端发送请求至收到响应之间的流逝时间。

响应时间测试和吞吐量测试（假设后者是基于客户端 – 服务器模式）之间的差别是，响应时间测试中的客户端线程会在操作之间休眠一段时间。这被称为思考时间。响应时间测试是尽量模拟用户行为：用户在浏览器输入 URL，用一些时间阅读返回的网页，然后点击页面上的链接，花一些时间阅读返回的网页，等等。

当测试中引入思考时间时，吞吐量就固定了：指定数量的客户端，在给定思考时间下总是得到相同的 TPS（少许差别，参见框注）。基于这点，测量请求的响应时间就变得重要了：服务器的效率取决于它响应固定负载有多快。

思考时间和吞吐量

有两种方法可以测试客户端包括思考时间时的吞吐量。最简单的方法就是客户端在请求之间休眠一段时间。

```
while (!done) {
    time = executeOperation();
    Thread.currentThread().sleep(30*1000);
}
```

这种情况下，吞吐量一定程度上依赖响应时间。如果响应时间是 1 秒，就意味着客户端每 31 秒发送一个请求，产生的吞吐量就是 0.032 OPS。如果响应时间是 2 秒，客户端就是每 32 秒发送一个请求，吞吐量就是 0.031 OPS。

另外一种方法是周期时间（Cycle Time）。周期时间设置请求之间的总时间为 30 秒，所以客户端休眠的时间依赖于响应时间：

```
while (!done) {
    time = executeOperation();
    Thread.currentThread().sleep(30*1000 - time);
}
```

无论响应时间是多少，这种方法都会产生固定的吞吐量，每个客户端 0.033 OPS（假设本例中的响应时间都少于 30 秒）。

测试工具中的思考时间时常有变，平均值为特定值，但会加入随机变化以更好地模拟用户行为。另外，线程调度从来不会严格实时，所以客户端请求之间的时间也会略有不同。

因此，即便工具提供周期时间而不是思考时间，测试所报告的吞吐量也相差无几。但是，如果吞吐量远超预期，说明测试中一定有什么出错了。

衡量响应时间有两种方法。响应时间可以报告为平均值：请求时间的总和除以请求数。响应时间也可以报告为百分位请求，例如第 90 百分位响应时间。如果 90% 的请求响应小于 1.5 秒，且 10% 的请求响应不小于 1.5 秒，则 1.5 秒就是第 90 百分位响应时间。

两种方法的一个区别在于，平均值会受离群值影响。这是因为计算平均值时包括了离群值。离群值越大，对平均响应时间的影响就会越大。

图 2-2 展示了 20 个请求，它们响应时间的范围比较典型。响应时间是从 1 到 5 秒。平均响应时间（平行且靠近 *x* 轴的粗线）为 2.35 秒，且 90% 的请求发生在 4 秒或 4 秒以内（平行且远离 *x* 轴的粗线）。

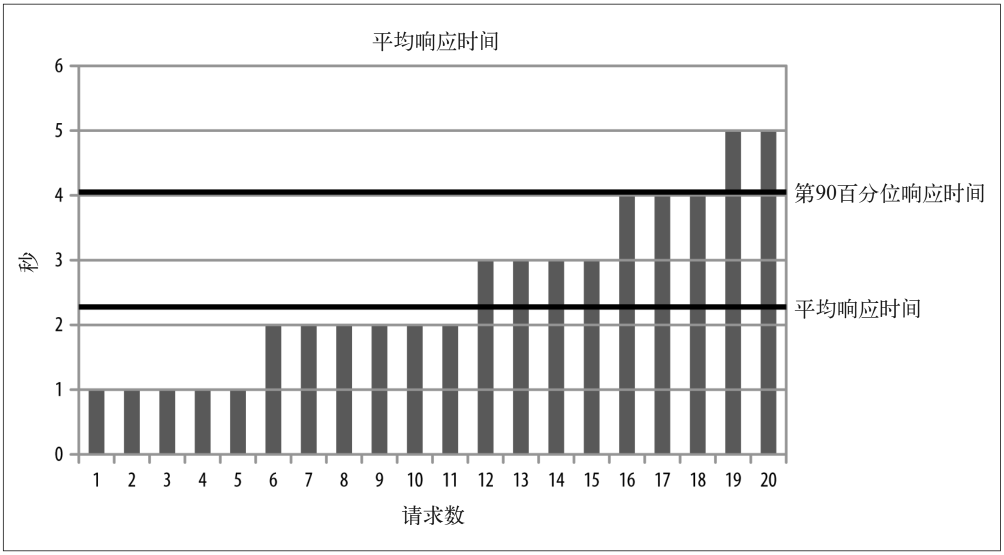


图 2-2：一组典型的响应时间

对于行为正常的测试来说，这是常见的场景。离群值会影响分析的准确性，就像图 2-3 显示的数据那样。

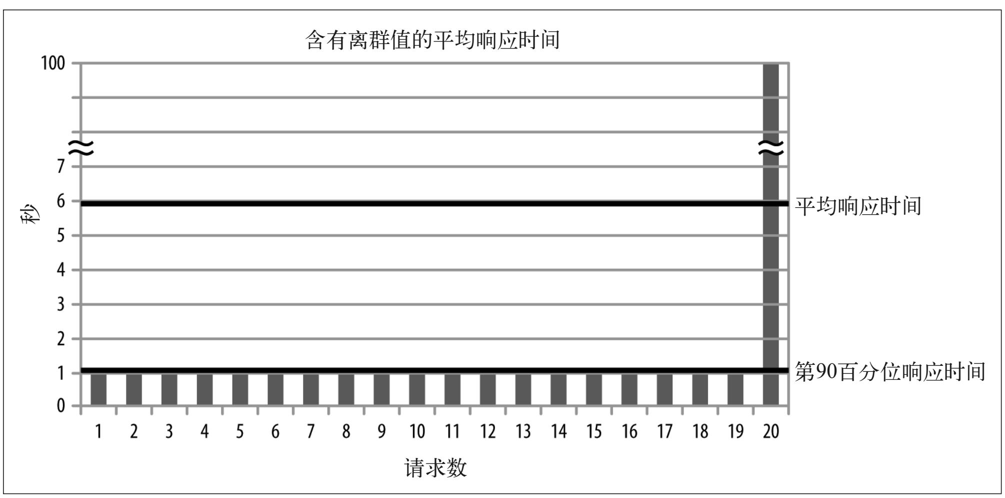


图 2-3：一组含有离群值的响应时间

这组数据中包括一个很大的离群值：有个请求花费了 100 秒。结果第 90 百分位响应时间和平均响应时间的粗线就调换了位置。平均响应时间蹿到了 5.95 秒，而第 90 百分位响应时间为 1.0 秒。对于这样的案例，应该考虑减少离群值带来的影响（从而降低平均响应时间）。

一般来说，像这样的离群值很少见，不过由于 GC（垃圾收集）引入的停顿，Java 应用更

容易发生这种情况。（并不是因为 GC 引入了 100 秒的延迟，而是尤其是对于有较小的平均响应时间的测试来说，GC 停顿会引入较大的离群值。）性能测试中通常关注的是第 90 百分位响应时间（有时是第 95 百分位或第 99 百分位响应时间，这里说第 90 百分位并没有什么神奇之处）。如果你只能盯住一个数字，那最好选择基于百分位数的响应时间，因为它的减少会让大多数用户受益。不过，最好一并考虑平均响应时间和至少一种百分位响应时间，你就不会错过有很大离群值的场景了。

负载生成器

有许多开源和商业的负载生成器。本书以 Faban（http://faban.org/）为例，这是一个开源的、基于 Java 的负载生成器。Faban 带有一个简单程序（fhb），可用来测试简单 URL 的性能：

```
% fhb -W 1000 -r 300/300/60 -c 25 http://host:port/StockServlet?stock=SDO
ops/sec: 8.247
% errors: 0.0
avg. time: 0.022
max time: 0.045
90th %: 0.030
95th %: 0.035
99th %: 0.035
```

这个测试例子中有 25 个客户端（`-c 25`）向 StockServlet 发送请求（股票代码 SDO），每个请求的周期时间为 1 秒（`-W 1000`）。`-r 300/300/60` 表示，基准测试的热身期为 5 分钟（300 秒），接下来是 5 分钟测试周期和 1 分钟减速期。测试之后，`fhb` 报告该测试的 OPS 和各种响应时间（由于包括思考时间，响应时间就成为重要的度量，而 OPS 则在不断变化）。

只要替换有限的几个参数，`fhb` 就可以处理 `POST` 数据，用少量脚本就可以处理多个 URL。对于更为复杂的测试来说，Faban 提供了很有用的 Java 框架来定义基准测试负载生成器。[4]

快速小结

1. Java 性能测试中很少使用面向批处理的测试（或者任何没有热身期的测试），但这种测试可以产生很有价值的结果。
2. 其他可以测量吞吐量或响应时间的测试，则依赖负载是否以固定的速率加载（也就是说，基于模拟的客户端思考时间）。

2.3 原则3：用统计方法应对性能的变化

第 3 条原则讲的是性能测试的结果会随时间而变。即便程序每次处理的数据集都相同，产生的结果也仍然会有差别。因为有很多因素会影响程序的运行，如机器上的后台进程，网

注 4：`fhb` 的命令行，请参见 http://faban.org/1.2/docs/man/fhb.html。——译者注

络时不时的拥堵等。好的基准测试不会每次都处理相同的数据集，而是会在测试中制造一些随机行为以模拟真实的世界。这就会带来一个问题：运行结果之间的差别，到底是因为性能有变化，还是因为测试的随机性。

可用以下方法来解决这个问题，即多次运行测试，然后对结果求平均。当被测代码发生变化时，就再多次运行测试，对结果求平均，然后比较两个平均值。这听起来似乎很容易。

不幸的是，事情并没有想象中那么简单。要想弄清楚测试间的差别是真实的性能变化还是随机变化并不容易——这就是性能调优的关键所在，不仅需要科学引领道路，还需要懂点艺术才能玩得转。

比较基准测试的结果时，我们不可能知道平均值的差异是真的性能有差还是随机涨落。最好的办法是先假设“平均值是一样的”，然后确定该命题为真时的几率。如果命题很大几率为假，我们就有信心认为平均值是真的有差别（虽然我们永远无法 100% 肯定）。

像这种因代码更改而进行的测试称为回归测试。在回归测试中，原先的代码称为基线（baseline），而新的代码称为试样（specimen）。考虑一个批处理程序的案例，基线和试样都执行 3 次，表 2-2 给出了所用的时间。

表2-2：假设两组测试的执行时间

迭代	基线（秒）	试样（秒）
第 1 次	1.0	0.5
第 2 次	0.8	1.25
第 3 次	1.2	0.5
平均值（秒）	**1**	**0.75**

试样的平均值表明代码性能改善了 25%。这个测试所反映出来的 25% 的改善，我们真的能相信多少？看上去很美好：试样中 3 个有 2 个的值小于基线平均值，看起来改进的步子很大——但是如果用本节介绍的方法分析这些结果就会得出结论，试样和基线性能相同的概率有 43%。观察到的这些数字说明，两组测试的基本性能在 43% 的时间内是相同的，因此性能不相同的时间只占 57%。顺便说一句，57% 的时间内性能不相同和性能改善 25% 也完全不是一回事，稍后讨论这个问题。

上述概率看起来和我们的预期有差别，其原因是测试的结果变化很大。一般来说，结果数据差别越大，就越难判断平均值之间的差异是真实的差别还是随机变动。

此处的数字 43%，是学生 t 检验（Student’s t-test，以下称 t 检验）得出的结果，这是一种针对一组数据及其变化的统计分析。顺便说一句，“学生”是首次发表该检验的科学家[5]的笔名，而不是提醒你（至少我）那些年在学校睡过的统计学课。t 检验计算出的 p 值，是指原假设（null hypothesis）成立时[6]的概率。（有一些程序和类库可以计算 t 检验，本节的结果是用 Apache Commons Mathematics 类库中的 `TTest` 计算的。）

回归测试中的原假设是指假设两组测试的性能一样。这个例子中的 p 值大约为 43%，意思

注 5：即威廉·戈斯特。——译者注

注 6：原文为“false”，有误。——译者注

是我们相信这两组测试平均值相同的概率为 43%。相反，我们相信平均值不同的概率为 57%。

57% 意味着什么，两组测试的平均值不相同？严格来讲，这并不意味着我们相信性能改善 25% 的概率有 57%——它只是意味着，我们相信结果不同的概率为 57%。性能可能改善了 25%，也可能 125%，甚至试样的实际性能也许比基线还糟糕。最大的可能则是测量出来的差别就是接近于真实的差异（特别是随着 p 值下降越是如此），只是我们永远无法肯定这点。

统计学及其语义

正确表述 t 检验结果的语句应该像这样：试样与基线有差别的可能性为 57%，差别预计最大有 25%。

不过通常会这么描述：结果改善 25% 的置信度（confidence level）为 57%。确切地说，这种说法与前面并不一致，也会让统计学家们抓狂，不过这种说法简短而易于为人接受，也不算太离谱。统计学分析经常会涉及不确定性，如果语义可以精确地陈述，自然能让人更好地理解这种不确定性。不过对于那些基础问题已经很清楚的领域，语义描述上有些悄然简化也是在所难免的。

t 检验通常与 α 值一起使用，α 值是一个点（有点随意），如果结果达到这个点那就是统计显著性（statistical significance）。通常 α 值设置为 0.1——意思是说，如果试样和基线只在 10%（0.1）的时间里相同（或反过来讲，90% 的时间里试样和基线有差异），那结果就被认为是统计显著。其他常用的 α 值还有 0.05（置信度为 95%）或 0.01（置信度为 99%）。如果测试的 p 值小于 1–α 值，则被认为是统计显著。

因此，查找代码性能变化的正确方法是先决定一个显著性水平——比如 0.1——然后用 t 检验判定在这个显著性水平上试样是否与基线有差别。请仔细搞明白，如果显著性测试失败，意味着什么。在这个例子中，p 值为 0.43，在置信度为 90% 的情况下我们不能说有显著性差异，而结果表示平均值不相同。事实上，测试没有显著性差异并不意味着结果无关紧要，它仅仅表示这个测试没法形成定论。

统计学中的显著性与重要性

显著性差异并不意味着统计结果对我们更重要。平均为 1 秒的变化很小的基线，和平均为 1.01 秒的变化很小的试样，其 p 值可能为 0.01：结果的差别有 99% 的置信度。

但结果的差别只有 1%。现在假定另外一个测试，试样和基线有 10% 的变动，但是 p 值为 0.2，即非统计显著。哪个测试的结果最为重要？这需要更多时间来审查。

审查后发现，虽然相差 10% 的测试的置信度低，但在用时上更加优化（如果可能的话，可以用更多数据来验证测试结果是否真的统计显著）。仅仅因为 1% 差异的可能性更大，并不意味着它更重要。

从统计学上说，测试不能得出定论通常是因为样本数据不足。迄今为止，示例所考虑的基

线和试样各是 3 次迭代。再加 3 次迭代，结果会变成这样：基线的迭代结果分别是 1、1.2 和 0.8 秒，试样的迭代结果分别是 0.5、1.25 和 0.5 秒？随着数据的增加，p 值就从 0.43 跌落到了 0.19，这意味着结果有差异的概率从 57% 上升到了 81%。运行更多测试迭代，再加 3 个数据点后，概率则增加到了 91%——超过了常规的统计显著性水平。

运行更多测试迭代从而达到某个统计显著性水平的方法并不总是可行。严格来说，这么做也没有必要。实际上，用以判定统计显著性的 α 值可以任意选择，虽然通常选的都是普遍认可的值。置信度为 90% 时，p 值 0.11 不是统计显著，但置信度为 89% 时，它就是统计显著了。

这里得到结论：回归测试并不是非黑即白的科学。对于一组数据，不经过统计分析你就没法弄清楚数字的含义，也就没法进行比较和判断。此外，由于概率的不确定性，即便用统计分析也不能给出完全可靠的结果。性能调优工程师的工作就是：考虑一堆数据（或者他们的均值）、弄清各种概率、决定往哪使力。

快速小结

1. 正确判定测试结果间的差异需要统计分析，通过统计分析才能确定这些差异是不是归因于随机因素。
2. 可以用严谨的 t 检验来比较测试结果，实现上述目的。
3. t 检验可以告知我们变动存在的概率，却无法告诉我们哪种变动该忽略，而哪种该追查。如何在两者之间找到平衡，是性能调优工程的艺术魅力所在。

2.4 原则4：尽早频繁测试

这是第 4 条也是最后的原则。性能极客们（包括我）喜欢将性能测试作为开发周期不可或缺的一部分。理想情况下，在代码提交到中心源代码仓库前，性能测试就应该作为过程的一部分运行，如果代码引入了性能衰减，提交就会被阻止。

本章中，建议之间有些内在的冲突，而建议和现实之间也有冲突。好的性能测试包含了许多代码——至少中等规模的介基准测试是这样。它需要在新老代码上重复运行多次，以便确认性能真的有差别而不是随机变动。在大型项目中，这可能需要花费好几天或者一周时间，这使得在提交代码到仓库之间运行性能测试变得不那么现实。

通常的软件开发周期也没使事情变得更容易。项目日程通常会固定特性的发布日期：所有的代码变动必须在发布周期的早些时候就提交到源代码仓库，而剩下的时间则贡献给了将新版本中的缺陷（包括性能问题）抖落干净。这导致了提早测试的两个问题。

(1) 为了赶上项目进度，开发人员会在时间压力之下提交代码，而一旦有时间修复性能问题时，又变得踯躅不前。早期提交代码所导致的 1% 的性能衰减，开发人员愿意承受压力，修复问题。而等到功能特性截止夜才提交的代码，如果性能衰减 20%，开发人员就只能以后再处理了。

(2) 代码发生变化，性能也会随之而变。这个道理与测试全应用（以及可能有的模块测试）相同：堆内存的使用情况会改变，代码编译也会改变，等等。

开发过程中无论有多少困难，频繁的性能测试仍然很重要，即便有时候不能立刻解决问题。比如代码性能衰减了 5%，随着开发的推进，开发人员或许可以采用以下措施：如果他的代码依赖有待集成的功能特性，那就等该功能可用时，再稍微调整代码，性能衰减的问题或许就解决了。这是合理的情况，即便这意味着性能测试不得不几个星期都伴随着 5% 的性能衰减（这是不幸的事，却又无法避免，还可能掩盖了其他问题）。

另一方面，如果性能衰减只有等架构更改才能修复的话，那就最好在其余代码开始依赖新代码实现之前，尽早捕获和解决它。这是一种平衡，需要仔细分析，甚至还常常需要点政治技巧。

遵循以下准则，可以使得尽早频繁测试变得最有用。

自动化一切

所有的性能测试都应该脚本化（或者程序化，虽然脚本更简单）。全部环境都必须通过脚本安装和配置新代码（创建数据库连接、建立用户账号等），然后用脚本运行测试集。所谓自动化，还不止这些：脚本必须能够多次运行测试，对结果进行 t 检验分析，并能生成置信度报告，说明统计结果是相同，还是不同，如果不同，相差多少。

在测试运行前，必须通过自动化技术确保机器处于已知状态：必须检查是否有不希望运行的进程，操作系统配置是否正确，等等。只有每轮运行时保持相同的环境，性能测试才是可重复的。自动化过程中必须考虑这点。

测试一切

必须自动收集能想象到的每一点数据，以便进行后续分析。这些数据包括整个运行过程中采集的系统信息：CPU 使用率、磁盘使用率、网络使用率和内存使用率等。数据还包括应用的日志——应用产生的日志，以及垃圾收集器的日志。理想情况下，还应该包括 JFR 记录的信息（参见第 3 章），或者对系统影响较小的性能分析（profiling）信息，周期性线程堆栈，以及其他堆分析数据，例如直方图或者全堆的转储信息（尤其是全堆转储，需要占用大量空间，没有必要长期保留）。

如果适用的话，监控信息还必须包括系统其他部分的数据：例如，如果程序使用数据库，就应该包括数据库机器的系统统计数据，以及所有的数据库诊断输出（包括 Oracle 的 Automatic Workload Repository [AWR] 这样的性能报告）。

这些数据可以指导所有未被覆盖的回归分析。如果 CPU 使用率上升，就需要参考性能分析信息，弄清楚是什么花费了这么多时间。如果 GC 时间变长，就该查阅堆性能分析信息，搞明白是什么消耗了这么多内存。如果 CPU 和 GC 时间都减少，某些地方的竞争可能降低了性能：栈数据可以指示特定的同步瓶颈（参见第 9 章），JFR 记录可用来发现应用的延迟，数据库日志也可以发现数据库竞争加剧的线索。

当发现性能衰减源时，需要进一步巡查，找到更多可用数据，更多可以追踪的线索。正如第 1 章所讨论的，发生性能衰减的未必是 JVM。测量一切，从而确保分析的正确性。

在真实系统上运行

在单核笔记本上运行测试，与在 256 线程 SPARC CPU 机器上有很大的不同。从线程效应上来说，原因很清楚：机器规模越大，同时能运行的线程就越多，从而能减少应用线

程对 CPU 的竞争。与此同时，大规模系统也会遇到小型笔记本上会被忽略的同步性能瓶颈。

还有其他重大的性能差异，即便乍一眼看上去不那么明显。许多重要的性能调优标志，它们的默认值是基于 JVM 运行的底层硬件系统计算出来的。平台和平台之间所编译出来的代码也不同。缓存——软件缓存以及更重要的硬件缓存——在不同系统和不同负载下也是不一样的，等等。

因此，除非在预期的负载和预期的硬件下测试，否则永远无法在测试中完全了解特定生产环境下的性能。可以在配置较低的硬件上运行规模较小的测试，以此来模拟和外推。在现实测试中，复制生产环境相当困难或昂贵。但外推只是简单的预测，即便在最好的情况下，预测也可能是错的。大规模系统远不只是将各部分加起来那么简单，没有什么测试能够代替在真实系统上的负载了。

快速小结

1. 虽然频繁的性能测试很重要，但并非毫无代价，在日常的开发周期中需要仔细斟酌。
2. 自动化测试系统可以收集所有机器和程序的全部统计数据，这可以为查找性能衰减问题提供必不可少的线索。

2.5 小结

性能测试包括各种权衡。面对诸多相互制约的选择，我们能否做出适当的决策，对于系统性能能否提升至关重要。

性能测试应该先测哪部分，与我们的经验和直觉息息相关。微基准测试在这方面的作用最小，它的用途仅限于为某些操作设立宽泛的指导。这为其他测试留下广泛的施展空间，从小模块的测试到大规模多层的应用环境。所有这些测试都有某方面的价值，如何选择就得依靠经验和直觉了。不过最终部署到生产环境中之后，除了全应用测试，就没什么可选了。只有到那时才能理解所有与性能相关的问题以及全部影响。

与此类似，哪些代码导致或没导致性能衰减，并不总是皂白分明的。程序时不时会表现出随机行为，而一旦引入了随机性，我们就再也无法 100% 确定这些数据意味着什么了。使用统计分析有助于使结果变得更客观，但即便如此，仍然免不了主观臆断。理解这些数据背后的概率及其意义，有助于降低主观性。

第 3 章

Java性能调优工具箱

性能分析过程中的一切都要能可视化，从而了解应用内部及应用所在的环境发生了什么。可视化的关键全在于工具，所以性能调优也完全在于工具。

在第 2 章中，我强调了用数据驱动性能调优的重要性：你必须对应用的性能进行测量，理解这些测量指标的含义是什么。性能分析应该与数据驱动的调优类似：为了让程序执行得更快，你必须掌握精确的运行数据。本章的主题就是如何获取并理解这些数据。

有许多工具可以提供 Java 应用的执行信息，当然全部介绍一遍是不现实的。最重要的工具多数都来自 JDK 或者开源站点 http://java.net。虽然还有其他开源和商业工具，但为方便起见，本章关注的主要是 JDK 所提供的工具。

3.1　操作系统的工具和分析

实际上性能分析的起点与 Java 无关：它是一组操作系统自带的基本监控工具。在基于 Unix 的系统上，有 `sar`（System Accounting Report）及其组成工具，例如 `vmstat`、`iostat`、`prstat` 等。在 Windows 上，有图形化资源监视器以及像 `typeperf` 这样的命令行工具。

无论何时运行性能测试，都应该收集操作系统的数据，至少需要收集 CPU、内存和磁盘使用率的信息。如果程序使用网络，还应该收集网络使用率。如果是自动化性能测试，还需要依靠命令行工具（即使是 Windows 系统）。不过，即便可以通过交互方式进行测试，也最好用命令行工具捕获输出，而不是一边盯着 GUI，一边琢磨它的意思。在分析的时候可以再次将这些输出图形化。

3.1.1　CPU使用率

我们先看 CPU 的监控，以及监控所能告诉我们的关于 Java 程序的信息。通常 CPU 使用率

可以分为两类：用户态时间和系统态时间（Windows 上被称作 privileged time）。用户态时间是 CPU 执行应用代码所占时间的百分比，而系统态时间则是 CPU 执行内核代码所占时间的百分比。系统态时间与应用相关，比如应用执行 I/O 操作，系统就会执行内核代码从磁盘读取文件，或者将缓冲数据发送到网络，等等。任何使用底层系统资源的操作，都会导致应用占用更多的系统态时间。

性能调优的目的是，在尽可能短的时间内让 CPU 使用率尽可能地高。这听起来有点不合常理。或许你此时正坐在电脑旁，看着它拼命挣扎，因为 CPU 使用率已经是 100% 了。好，我们先来考虑一下，CPU 使用率到底反映了什么。

首先需要注意的是，CPU 使用率是一段时间内的平均数——5 秒、30 秒，也可能只有 1 秒那么短（不过永远不会比这还要短）。比如，10 分钟内一个程序执行的 CPU 使用率为 50%。如果代码调优之后，CPU 使用率达到了 100%，说明程序的性能翻了倍：程序只需要执行 5 分钟就可以了。如果性能再翻倍，CPU 仍将是 100%，而执行完程序只要 2.5 分钟。CPU 使用率表示程序以多高的效率使用 CPU，所以数字越大，性能越好。

如果在 Linux 桌面系统上运行 `vmstat 1`，可以得到类似如下的几行信息（每隔一秒显示一行）：

```
% vmstat 1
procs -----------memory---------- ---swap-- -----io---- -system-- ----cpu----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa
 2  0      0 1797836 1229068 1508276 0    0     0     9 2250 3634 41  3 55  0
 2  0      0 1801772 1229076 1508284 0    0     0     8 2304 3683 43  3 54  0
 1  0      0 1813552 1229084 1508284 0    0     3    22 2354 3896 42  3 55  0
 1  0      0 1819628 1229092 1508292 0    0     0    84 2418 3998 43  2 55  0
```

为了便于说明问题，这个运行示例程序只有一个活跃线程。不过即便有多个线程，也可以应用如下概念。

每秒内，CPU 被占用 450 毫秒（42% 的时间执行用户代码，3% 的时间执行系统代码）。相应地，CPU 空闲 550 毫秒。CPU 空闲可能有以下原因。

- 应用被同步原语阻塞，直至锁释放才能继续执行。
- 应用在等待某些东西，例如数据库调用所返回的响应。
- 应用的确是无所事事。

前面 2 种情况通常都可用来识别某些问题。如果竞争降低，或优化数据库使之发送响应更快，程序运行都能变得更快，平均 CPU 使用率也会上升（当然，得假定没有其他继续阻塞应用的问题）。

第 3 点则常常使人疑惑。如果应用有事情做（而不是因为等待锁或者其他资源而无事可干），CPU 就会分配一些周期执行应用代码。这是一般性原则，并不只针对 Java。比如，包含无限循环的简单脚本。这段脚本执行时，将消耗 100% 的 CPU。以下的 Windows 批处理任务就是这么干的：

```
ECHO OFF
:BEGIN
ECHO LOOPING
```

```
GOTO BEGIN
REM We never get here…
ECHO DONE
```

考虑一下，如果这段脚本没有消耗 100%CPU，那意味着什么。意味着，操作系统还有些事可做——它可以打印一行 LOOPING——却选择了空闲。这种情况下，空闲并没有什么好处，如果我们正在进行一些有用（耗时）的计算，那么迫使 CPU 周期性空闲只会使我们得到响应的时间变得更长。

如果在单 CPU 机器上运行上述脚本，多数时候你不会注意到它的运行。不过一旦开启新程序，或者测量其他程序的运行时间，你就能看到影响了。操作系统擅长为争用 CPU 周期的程序分配时间片，但新程序可用的 CPU 变少了，它也就运行得更慢。所以基于这种经验，人们有时会认为，在其他程序可能需要 CPU 周期时预留一些空闲周期，没准是个好主意。

但操作系统无法猜到你接下来想做什么，所以（默认情况下）它会尽可能执行一切而不是让 CPU 空闲。

限制程序所用的 CPU

尽可能利用 CPU 周期运行程序可以使程序性能最大化。不过有时你并不希望如此。比如，你运行 *SETI@home*[1]，它将消耗你机器所有可用的 CPU 周期。这在你不干活的时候没事，上网冲浪或者写文档的时候也没事，否则就会降低你的生产率。（不妨考虑一下如果你正在玩 CPU 密集型游戏，这样做会发生什么。）

操作系统有许多机制可用来人为限定程序所使用的 CPU——事实上，如果有程序需要使用 CPU，它就会退出空闲周期。进程的优先级也可以改变，所以那些后台任务既不会与你想运行的程序争用 CPU，也不会让 CPU 处于空闲状态。这些技术超出了我们讨论的范围。郑重说一句，SETI@home 可以让你配置优先级，除非你允许，否则它不会真的占用你机器的所有空余周期。

1. Java和单CPU的使用率

再回来讨论 Java 应用——CPU 周期性空闲意味着什么？这依赖于应用的类型。如果应用代码是批处理类型，工作量固定，你应该永远都不会看到 CPU 空闲，因为这意味着没事可做。提高 CPU 使用率，一直都是批处理任务的目的，因为任务会很快完成。如果 CPU 已经达到 100%，你仍然可以寻找优化，使得工作完成的更快（也要尽量保持 100%CPU 使用率）。

如果测试接收请求的服务器应用，就可能出现因无事可做而出现的空闲：例如，Web 服务器已经处理完所有未完成的 HTTP 请求，正在等待下一个请求的时候。这就引入了平均时间。上述 vmstat 的示例来自一个每秒接收一个请求的应用服务器。应用服务器花 450 毫秒处理请求——意思是 CPU 被 100% 占用 450 毫秒，550 毫秒没有占用。这就是所报告的 CPU 被占用 45%。

注 1：这是利用全球联网计算机搜寻外星文明的科学实验。——译者注

虽然经常因为 CPU 占用发生的时间粒度很小而难以可视化，但运行负载型应用时 CPU 的行为就是这种爆发式的。如果 CPU 每半秒收到一个请求而平均处理时间为 225 毫秒，也能从宏观层面看到同样的模式。CPU 被占用 225 毫秒，空闲 275 毫秒，再占用 225 毫秒，空闲 275 毫秒：平均来看，被占用 45%，空闲 55%。

如果应用优化之后每个请求只需要 400 毫秒，整体 CPU 使用率就会减少（到 40%）。这是仅有的降低 CPU 使用率有意义的情况——当系统负载量固定并且应用不受外部资源限制的时候。另一方面，优化也使系统可以承担更多负载，最终提高 CPU 使用率。微观来看，这种情况下的优化仍然是使 CPU 使用率在短时间内（执行请求花费 400 毫秒）变为 100%——只是 CPU 峰值持续的时间很短，事实上，大多数工具都不会将其标记为 100%。

2. Java和多CPU的使用率

上面的例子是假定在单个 CPU 上运行的单线程，但概念与一般情况下多 CPU 多线程相同。多线程倾向于以有趣的方式平均使用 CPU——第 5 章有这样的例子，展示多个 GC 线程如何使用 CPU。但一般来说，多 CPU 多线程的目的仍然是通过不阻塞线程来提高 CPU 使用率，或者是在线程完成工作等待更多任务时降低 CPU 使用率。

另外在多线程多 CPU 下，需要重点考虑以下 CPU 空闲的情形：即便有事可做，CPU 仍然空闲。这在程序没有更多线程可用的时候可能会出现。典型的情况是，应用以固定尺寸的线程池运行各种任务。每个线程同时只能执行一个任务，当线程被某个任务阻塞时（例如，等待数据库的响应），它就没法捡出新任务执行了。所以此时的情况就是，有任务需要执行（有事可做），却没有线程执行它们，结果就是 CPU 处在空闲时间。

在这个例子中，应该增加线程池的大小。然而，不要假设所有的空闲都是因为 CPU 可用，从而增加线程池以完成更多工作。程序得不到 CPU 周期，还有可能是由于前面提到的两个原因——锁或者外部资源的瓶颈。很重要的一点就是，在决定采取行动前，需要搞清楚为什么程序得不到 CPU。（这个主题的更多详细内容请参见第 9 章。）

检查 CPU 使用率是弄清楚应用性能的第一步，但它的用途不仅如此：它可以查看代码的 CPU 使用是否与预期的一样，或者可以指出一些同步或资源问题。

3.1.2 CPU运行队列

Windows 和 Unix 系统都可以监控可运行（意味着没有被 I/O 阻塞、休眠等）的线程数。Unix 系统称之为*运行队列*（run queue），一些工具的输出中也有运行队列长度。上节 `vmstat` 的输出中就包括：每行的首个数字就是运行队列长度。Windows 将这个数字称为*处理器队列*（processor quque），`typeperf`（还有其他方法）可以报告该信息：

```
C:> typeperf -si 1 "\System\Processor Queue Length"
"05/11/2013 19:09:42.678","0.000000"
"05/11/2013 19:09:43.678","0.000000"
```

前面 `vmstat` 的输出与此处有很重要的差别：Unix 系统的运行队列长度（`vmstat` 输出示例中的 1 或 2）是所有*正在运行*或*待运行*（即一旦有可用 CPU 就可以运行）的线程数。示例中至少有一个线程试图运行：即以单线程执行应用。因此，运行队列长度至少是 1。记住，运行队列反映的是机器上所有东西的运行情况，所以示例输出中有时会看到运行队列长度

为 2，因为此时有其他线程（来自其他完全隔离的进程）试图运行。

而在 Windows 中，处理器队列长度就不包括正在运行的线程。因此，上述 typeperf 示例输出中的处理器队列长度就是 0，即便机器上同样有一个单线程应用的线程一直在执行。

如果试图运行的线程数超过了可用的 CPU，性能就会下降。一般来说，Windows 的处理器队列长度最好为 0，小于或等于 Unix 系统 CPU 的数目。不过，这不是硬性的规定。有些系统或其他进程会周期性出现，在这瞬间数字会有提高，这对性能不会有实质性的影响。但是，如果在相当长时间内运行队列很长，说明系统已经过载，这时你应该检查系统，减少机器正在处理的工作量（将工作转移到其他机器或者优化代码）。

快速小结

1. 检查应用性能时，首先应该审查 CPU 时间。
2. 优化代码的目的是提升而不是降低（更短时间段内的）CPU 使用率。
3. 在试图深入优化应用前，应该先弄清楚为何 CPU 使用率低。

3.1.3　磁盘使用率

监控磁盘使用率有两个目的。第一个目的与应用本身有关：如果应用正在做大量的磁盘 I/O 操作，那 I/O 就很容易成为瓶颈。

想了解何时磁盘 I/O 是瓶颈非常困难，因为这取决于应用的行为。如果应用往磁盘写数据时（例子参见第 12 章）没有有效的缓冲，磁盘 I/O 的统计数据就会非常低。但是，如果应用执行的 I/O 超过了磁盘的承载，磁盘 I/O 的统计数据就会非常高。请注意，这两种情形的性能都需要提升。

有些系统的基本 I/O 监控要好于其他系统。这是 Linux 系统 iostat 的部分输出：

```
% iostat -xm 5
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          23.45    0.00   37.89    0.10    0.00   38.56

          Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s
          sda               0.00    11.60    0.60   24.20     0.02

          wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
          0.14    13.35     0.15    6.06    5.33    6.08   0.42   1.04
```

应用正在往磁盘 sda 写数据。乍一看，磁盘统计数据还不错。w_await——每次 I/O 写的时间——相当低（6.08 毫秒），磁盘使用率只有 1.04%。（可接受的值取决于物理磁盘，在低于 15 毫秒时，我的台式机系统 5200 RPM 的磁盘可以工作得很好。）但这里有条线索可以看出点问题：系统在内核花费了 37.89% 的时间。一种可能是系统正在进行其他 I/O（在其他程序中）。如果这个系统时间都来自被测的应用，说明某些低效率的事正在发生。

另一条线索是，系统每秒写为 24.2：当每秒写入只有 0.14 MB 时，这算很大的数字。这说明 I/O 已经是瓶颈，接下来应该检查应用是如何写的。

如果磁盘速度赶不上 I/O 请求，问题的另外一面就出现了：

```
% iostat -xm 5
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          35.05    0.00    7.85   47.89    0.00    9.20

          Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s
          sda               0.00     0.20    1.00  163.40     0.00

          wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
          81.09  1010.19   142.74  866.47   97.60  871.17   6.08 100.00
```

Linux 好处在于可以立即告诉我们磁盘的使用率为 100%。它也能告诉我们进程的 47.89% 的事件在 iowait（表示正在等待磁盘）。

即便其他平台只能提供原始数据，也可以告诉我们哪里出错了：完成 I/O（w_await）用时 871 毫秒，队列很长，磁盘每秒写入 81 MB 数据。所有这些都说明磁盘 I/O 有问题，应用（或者可能是系统的其他部分）的 I/O 必须要降低。

监控磁盘使用率的第二个理由是——即便预计应用不会有很高的 I/O——有助于监控系统是否在进行内存交换。计算机的物理内存量是固定的，但它们可以用大得多的虚拟内存来运行一系列应用。应用会保留更多超过它们实际所需的内存量，并且它们通常也只使用分配给它们的内存的一部分。这两种情况下，操作系统可以将不用的内存保留在磁盘上，在需要时换页到物理内存。

大多数情况下，这种类型的内存管理可以工作得很好，特别是交互式应用和 GUI 程序（这一点是很有好处的，否则你的笔记本就需要比实际多得多的内存）。这种管理方式对服务器类应用来说效果稍差，因为这些应用需要更多内存。由于 Java 堆的原因，这种管理方式对于任何 Java 程序（包括你桌面上运行的基于）来说都比较糟糕。更多详情请参见第 5 章。

正在内存交换的系统——从主内存移动数据到磁盘或者反过来——一般来说，性能比较差。还有其他系统工具可以报告系统交换，例如 vmstat 输出中有两列（si 是换进，so 是换出）可以警告我们系统是否正在交换。磁盘活动说明内存交换可能正在发生。

快速小结

1. 对于所有应用来说，监控磁盘使用率非常重要。即便不直接写磁盘的应用，系统交换仍然会影响它们的性能。
2. 写入磁盘的应用遇到瓶颈，是因为写入数据的效率不高（吞吐量太低），或者是因为写入太多数据（吞吐量太高）。

3.1.4　网络使用率

如果应用运行时需要网络——比如 Java EE 应用服务器——你也必须监控网络流量。网络使用率类似磁盘流量：应用可能没有充分利用网络所以带宽很低，或者写入某网络接口的总数据量超过了它所能处理的量。

不幸的是，由于标准的系统工具通常只能显示某个网络接口发送和接收的数据报数和字节数，所以它们在监控网络流量方面差强人意。虽然这些信息有用，但无法告诉我们网络是

没有充分利用，还是过度使用。

Unix 系统监控网络的基本工具是 netstat（大多数 Linux 发行版中还没有包括 netstat，必须单独获得）。Windows 上则可以在脚本中使用 typeperf，监控网络使用率——不过 GUI 的优势可以显现出来的，标准的 Windows 资源监视器显示网络使用百分比的图。不幸的是，GUI 在自动性能测试场景中几乎没有什么帮助。

幸运的是，有许多开源和商业工具可以监控网络带宽。Unix 里一个受欢迎的命令行工具就是 nicstat，它可以显示每个网络接口的流量概要，包括网络接口的使用度：

```
% nicstat 5
Time      Int       rKB/s   wKB/s   rPk/s   wPk/s   rAvs    wAvs    %Util   Sat
17:05:17  e1000g1   225.7   176.2   905.0   922.5   255.4   195.6   0.33    0.00
```

示例中的 e1000g1 是 1000 MB 接口，使用率非常低（0.33%）。这个工具（以及其他类似的工具）可以用来计算接口的使用率。在上述输出中，接口的数据写入速率是 225.7 Kbps，读取速率是 176.2 Kbps。对于 1000 MB 的网络，相除以后可以得到使用率 0.33%，nicstat 也能自动算出接口的带宽。

typeperf 或 netstat 这样的工具可以报告读取和写入的数据，但是要计算网络使用率，你必须自己用脚本计算接口的带宽。虽然一般工具报告的单位是字节 / 秒（Bps），但请切记，带宽的单位是位 / 秒（bps）。1000 兆位网络每秒处理 125 兆字节（MB）。本示例中，读为 0.22 MBps，写为 0.16 MBps，相加然后除以 125 得出使用率为 0.33%。所以 nicstat（或类似工具）没有什么神奇的，只是更便于使用而已。

网络无法支持 100% 的使用率。对本地以太局域网来说，承受的网络使用率超过 40% 就意味着接口饱和了。如果网络是包交换或使用不同的传输介质，网络使用率的最大值就可能会不同，因此最好是评估网络架构之后再确定合适的值。这个值与 Java 无关，只是简单利用网络参数和操作系统接口。

快速小结

1. 对基于网络的应用来说，务必要监控网络以确保它不是瓶颈。
2. 往网络写数据的应用遇到瓶颈，可能是因为写数据的效率太低（吞吐量太低），也可能是因为写入了太多的数据（吞吐量太高）。

3.2 Java监控工具

要想深入了解 JVM 自身，需要使用 Java 的监控工具。JDK 自带以下所列工具。

- jcmd

  它用来打印 Java 进程所涉及的基本类、线程和 VM 信息。它适用于脚本，可以像这样执行：

  ```
  % jcmd process_id command optional_arguments
  ```

  jcmd help 可以列出所有的命令。jcmd help <command> 可以给出特定命令的语法。

- `jconsole`

  提供 JVM 活动的图形化视图，包括线程的使用、类的使用和 GC 活动。
- `jhat`

  读取内存堆转储，并有助于分析。这是事后使用的工具。
- `jmap`

  提供堆转储和其他 JVM 内存使用的信息。可以适用于脚本，但堆转储必须在事后分析工具中使用。
- `jinfo`

  查看 JVM 的系统属性，可以动态设置一些系统属性。可适用于脚本。
- `jstack`

  转储 Java 进程的栈信息。可适用于脚本。
- `jstat`

  提供 GC 和类装载活动的信息。可适用于脚本。
- `jvisualvm`

  监视 JVM 的 GUI 工具，可用来剖析运行的应用，分析 JVM 堆转储（事后活动，虽然 `jvisualvm` 也可以实时抓取程序的堆转储）。

这些工具可广泛用于以下领域：

- 基本的 VM 信息
- 线程信息
- 类信息
- 实时 GC 分析
- 堆转储的事后处理
- JVM 的性能分析

你可能注意到了，工具和适用领域并非一一对应的，许多工具可用于多个领域。所以我们不是单个研究每个工具，而是着眼于 Java 重要的可观测领域，讨论这些工具如何提供这类信息。同时，我们还会讨论其他工具（有些是开源，有些是商业），虽然提供的基本功能相同，但是相比基本的 JDK 工具具有一定的优势。

3.2.1　基本的VM信息

JVM 工具可以提供 JVM 进程的基本运行信息：它运行多久了，使用哪些 JVM 标志，以及 JVM 的系统属性，等等。

运行时间

此命令可以查看 JVM 运行的时长：

```
% jcmd process_id VM.uptime
```

系统属性

以下命令可以显示 System.getProperties() 的各个条目。

```
% jcmd process_id VM.system_properties
```

或者

```
% jinfo -sysprops process_id
```

这包括通过命令行 -D 标志设置的所有属性，应用动态添加的所有属性和 JVM 的默认属性。

JVM 版本

用以下方式获取 JVM 版本：

```
% jcmd process_id VM.version
```

JVM 命令行

jconsole 的“VM 摘要”页可以显示程序所用的命令行，或者用 jcmd 显示：

```
% jcmd process_id VM.command_line
```

JVM 调优标志

可用以下方式获得对应用生效的 JVM 调优标志：

```
% jcmd process_id VM.flags [-all]
```

调优标志

JVM 可以设置许多调优标志，本书就关注了其中很多标志。追踪这些标志及其默认值有点让人崩溃。上面最后两个 jcmd 示例对于获取这类信息很有用。command_line 显示直接在命令行指定的标志。flags 显示命令行设置的标志，以及 JVM 直接设置的标志（因为它们的值是通过自动优化决定的）。该命令加上 all 时，可以列出 JVM 内部所有的标志。

有几百个 JVM 调优标志，大多数都很令人费解，而且我们建议永远都不要对其作更改（参见下文的框注“信息太多？”）。诊断性能问题时，找出哪些标志起作用是很常见的事。JVM 运行时，可以用 jcmd 做到这一点。如果想找出特定 JVM 的平台特定的默认值是什么，那么在命令行上添加 -XX:+Printflagsfinal 会很有用。

想知道特定平台所设置的标志是什么，可以执行以下命令：

```
% java other_options -XX:+PrintFlagsFinal -version
……几百行输出,包括……
uintx InitialHeapSize                          := 4169431040    {product}
intx InlineSmallCode                           = 2000           {pd product}
```

你应该在命令行包括所有标志，因为有些标志会影响其他标志，特别是 GC 相关的标志。这个命令会打印 JVM 标志及其取值的完整列表（结果和 jcmd 结合 VM.flags -all 打印的相同）。

这些命令的标志数据以上述两种方式之一显示。输出第 1 行中的冒号表示标志使用的是非默认值。发生这种情况，可能是以下原因导致。

(1) 标志值直接在命令行指定。

(2) 其他标志间接改变了该标志的值。

(3) JVM 自动优化计算出来的默认值。

第 2 行（没有冒号）表示，值是这个 JVM 版本的默认值。某些标志的默认值在不同平台上可能会不相同，输出的最右列会指示。product 表示在所有平台上的默认设置都是一致的。pd product 表示标志的默认值是独立于平台的。

> **信息太多？**
>
> PrintFlagsFinal 会打出好几百个 JVM 可用的调优参数（比如 JDK 7u40 有 668 个可能的标志）。
>
> 绝大多数标志的存在都是为了支持工程师收集更多与应用运行（以及错误行为）相关的信息。有个比较吸引人的标志称为 AllocatePrefetchLines（默认值为 3），它使得标志值可以改变，从而使预读指令在特定处理器上可以工作得更好。但这种随意的调优并没有必要。除非你有很充分的理由，否则不要更改标志值。就 AllocatePrefetchLines 标志来说，需要掌握应用预读性能、运行应用的 CPU 的特性以及更改这个数字对 JVM 自身的代码有什么影响。

最后一列可能的值还有 manageable（运行时可以动态更改标志的值）和 C2 diagnostic（为编译器工程师提供诊断输出，帮助理解编译器正以什么方式运作）。

还有另一种查看运行中的应用的此类信息的工具，叫作 jinfo。jinfo 的好处在于，它允许程序在执行时更改某个标志的值。

以下是如何获取进程中所有标志的值：

```
% jinfo -flags process_id
```

jinfo 带有 -flags 时可以提供所有标志的信息，否则只打印命令行所指定的标志。这两种数据都不像 -XX:+Printflagsfinal 那样易读，但 jinfo 有其他值得注意的特性。

jinfo 可以检查单个标志的值：

```
% jinfo -flag PrintGCDetails process_id
-XX:+PrintGCDetails
```

虽然 jinfo 本身不会显示是否 manageable，但 manageable（如 Printflagsfinal 输出中所标识的）的标志可以通过 jinfo 开启或关闭：

```
% jinfo -flag -PrintGCDetails process_id # turns off PrintGCDetails
% jinfo -flag PrintGCDetails process_id
-XX:-PrintGCDetails
```

需要当心的是，jinfo 可以更改任意标志的值，但并不意味着 JVM 会响应更改。比如说，大多数影响 GC 算法行为的标志都在启动时使用，以决定垃圾收集器的行为方式。之后通过 jinfo 更改标志值，并不会导致 JVM 改变它的行为。它会以初始时的算法继续执行。所以这个技术只会对那些在 Printflagsfinal 输出中标记为 manageable 的标志有效。

快速小结

1. jcmd 可用来查找运行中的应用所在 JVM 的基本信息——包括所有调优标志的值。
2. 命令行上添加 -XX:+Printflagsfinal 可输出标志的默认值。这在查看特定平台自动优化所判定的默认值时很有用。
3. jinfo 在检查（某些情况下可以更改）单个标志时很有用。

3.2.2　线程信息

jconsole 和 jvisualvm 可以实时显示应用中运行的线程的数量。

查看运行线程的栈信息，对于判断线程是否被阻塞很有用。可以通过 jstack 获取栈信息：

```
% jstack process_id
…… 显示了每个线程的栈的众多输出 ……
```

也可以通过 jcmd 获取栈信息：

```
% jcmd process_id Thread.print
…… 显示了每个线程的栈的众多输出 ……
```

监控线程栈的详情请参见第 9 章。

3.2.3　类信息

jconsole 或 jstat 可以提供应用已使用类的个数。jstat 还能提供类编译相关的信息。

应用使用类的更多细节请参见第 12 章，监控类编译的细节请参见第 4 章。

3.2.4　实时GC分析

几乎所有的监控工具都能报告一些 GC 活动的信息。jconsole 可以用实时图显示堆的使用情况。jcmd 可以执行 GC 操作。jmap 可以打印堆的概况、永久代信息或者创建堆转储。jstat 可以为垃圾收集器正在执行的操作生成许多视图。

想了解这些程序如何监控 GC 活动，请参见第 5 章的实例。

3.2.5　事后堆转储

jvisualvm 的 GUI 界面可以捕获堆转储，也可以用命令行 jcmd 或 jmap 生成。堆转储是堆使用情况的快照，可以用不同的工具进行分析，包括 jvisualvm 和 jhat。传统上，第三方处理堆转储的工具都领先于 JDK，所以第 7 章使用第三方工具——Eclipse Memory Analyzer Tool——为例，展示如何在事后处理堆转储。

3.3　性能分析工具

性能分析器是性能分析师工具箱中最重要的工具。Java 有许多性能分析器，各有优缺点。

性能分析经常需要使用各种工具——特别是那些采样分析器。即便是相同的应用，不同的分析器也能发现其他分析器所发现不了的问题。

几乎所有的 Java 性能分析工具都是用 Java 写的，并以“关联”（attaching）应用的方式进行性能分析——意思是性能分析器开启与目标应用之间的 socket（或其他通信通道）。随后目标应用和性能分析工具交换应用的行为信息。

这意味着，就像调优任何其他 Java 应用一样，你必须注意性能分析工具自身性能的调优。尤其是如果应用很大，需要传递给分析工具的数据非常多，那么分析工具就必须得有足够大的堆内存来处理这些数据。性能分析工具采用并发 GC 算法运行，通常是个不错的主意。在性能分析过程中，不合时宜的 Full GC 停顿会导致性能分析工具缓冲区中的数据溢出。

3.3.1 采样分析器

性能分析有两种模式：数据采样或数据探查。数据采样是性能分析的基本模式，带来的开销最小，这点很重要。性能分析为人诟病的一点就是，对应用进行的测量会改变它的性能。（然而，你必须进行性能分析：还有什么方法能让你知道程序中的猫是否存活？）限制性使用性能分析可以使得测试结果更接近现实模型，即通常情况下应用的表现行为。

不幸的是，采样分析器可能会遇到各种错误。计时器定期触发采样分析器，然后采样分析器检查每个线程并判断正在执行的方法。

图 3-1 是采样中最常见的错误。线程交替执行 `methodA`（见图中的阴影块）和 `methodB`（见图中的白色块）。如果计时器只在线程执行 `methodB` 时触发，采样分析器就会报告成，在所有这段时间内线程都在执行 `methodB`，而实际上更多时间在执行 `methodA`。

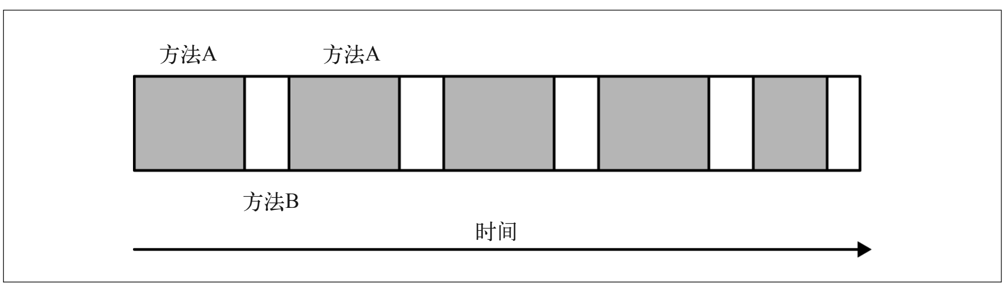
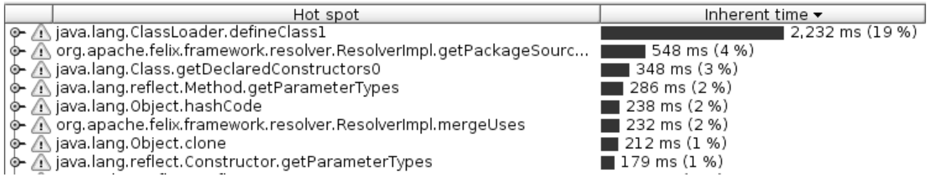


图 3-1：交替方法执行

常见的采样错误还不止这一个。减少这类错误的方法是，拉长分析的时间段，同时减少采样间隔。但是减少采样间隔和尽量减小性能分析影响应用的目的相违背，这里需要有平衡。不同的性能分析工具考虑的平衡点也不同，这就是为什么不同的性能分析工具所报告的数据有很大差别的原因。

图 3-2 是 GlassFish 应用服务器启动一个域时所测量出的基本采样分析。图中显示，大块时间（19%）都用在了 `defineClass1()` 上，接着是 `getPackageSourcesInternal()` 等方法。程序的启动性能是受 JVM 定义类过程的控制的，这并不奇怪。为了加快代码运行速度，必须改进类加载的性能。



Hot spot	Inherent time ▾
java.lang.ClassLoader.defineClass1	2,232 ms (19 %)
org.apache.felix.framework.resolver.ResolverImpl.getPackageSourc...	548 ms (4 %)
java.lang.Class.getDeclaredConstructors0	348 ms (3 %)
java.lang.reflect.Method.getParameterTypes	286 ms (2 %)
java.lang.Object.hashCode	238 ms (2 %)
org.apache.felix.framework.resolver.ResolverImpl.mergeUses	232 ms (2 %)
java.lang.Object.clone	212 ms (1 %)
java.lang.reflect.Constructor.getParameterTypes	179 ms (1 %)

图 3-2：基于一个样例的测试

请仔细留意上面的最后一句话：必须改善类加载的性能，而不是改善 `defineClass1()` 的性能。改善性能的通常设想是，应该先优化性能分析结果中排在最上面的方法。然而，这种做法常常受限。这个案例中的 `defineClass1()` 是 JDK 的一部分，而且是本地方法（native method），除非改写 JVM，否则不可能改善它的性能。而且，即便能改写 JVM 使得这个方法少花 60% 的时间，但换算成整体性能的改善，也不过 10%——这无异于杯水车薪。

而更常见的情况是，排在性能分析结果顶上的方法只占了整体时间的 2% 或 3%。即便将它所用的时间砍掉一半（通常极为困难），也只会使应用的性能提高 1%。只盯着性能分析结果中最上头的方法，通常并不会让性能提高很多。

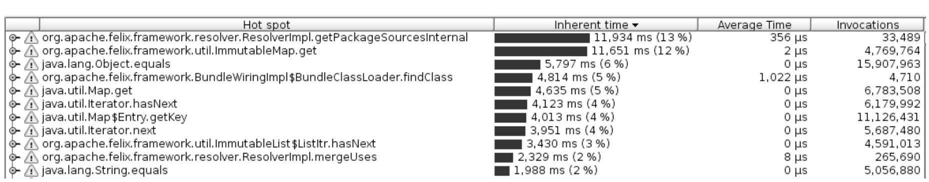
相反，你应该在最顶上那些方法所指引的区域中搜寻可优化的地方。GlassFish 性能工程师不会试图让类定义更快，但通常他们会找出如何加速类加载——装载更少的类、并行加载类等。

快速小结

1. 采样分析器是最常用的分析器。
2. 因为采样分析器的采样频率相对较低，所以引入的测量失真也较小。
3. 不同的采样分析器各有千秋，针对不同应用各有所长。

3.3.2　探查分析器

探查分析器相比于采样分析器，侵入性更强，但它们可以给出关于程序内部所发生的更有价值的信息。图 3-3 探查的是与图 3-2 相同的 GlassFish 域，采用的也是同一个性能分析工具，但这次使用的是探查模式。



Hot spot	Inherent time ▾	Average Time	Invocations
org.apache.felix.framework.resolver.ResolverImpl.getPackageSourcesInternal	11,934 ms (13 %)	356 µs	33,489
org.apache.felix.framework.util.ImmutableMap.get	11,651 ms (12 %)	2 µs	4,769,764
java.lang.Object.equals	5,797 ms (6 %)	0 µs	15,907,963
org.apache.felix.framework.BundleWiringImpl$BundleClassLoader.findClass	4,814 ms (5 %)	1,022 µs	4,710
java.util.Map.get	4,635 ms (5 %)	0 µs	6,783,508
java.util.Iterator.hasNext	4,123 ms (4 %)	0 µs	6,179,992
java.util.Map$Entry.getKey	4,013 ms (4 %)	0 µs	11,126,431
java.util.Iterator.next	3,951 ms (4 %)	0 µs	5,687,480
org.apache.felix.framework.util.ImmutableList$ListItr.hasNext	3,430 ms (3 %)	0 µs	4,591,013
org.apache.felix.framework.resolver.ResolverImpl.mergeUses	2,329 ms (2 %)	8 µs	265,690
java.lang.String.equals	1,988 ms (2 %)	0 µs	5,056,880

图 3-3：一个探查分析器

我们立即就能从这份性能分析结果中看出些问题。首先，现在最“热”的方法成了 getPackageSourcesInternal()，因为占用时间达到了 13%（而不是前个例子中的 4%）。性能分析结果中还有其他几个占用很多时间的方法，也都排到了前头，而 defineClass1() 已经完全消失了。这次工具还报告了每个方法被调用的次数，并且基于这个次数计算出了每次调用的平均时间。

难道这个性能分析结果比采样的好？这取决于能否有办法知道在给定的情况下哪种分析结果更精确。探查分析结果中的调用次数毫无疑问是精确的，而其他信息在判断哪段代码实际花费了更多时间，哪些有更多的优化空间时通常很有帮助。在这个例子中，虽然 ImmutableMap.get() 消耗了 12% 的整体时间，但它被调用了约 470 万次。减少这个方法的总调用次数比加快它的运行速度可以更显著地提升性能。

不过话说回来，探查分析器会在类加载时更改类的字节码（即插入统计调用次数的代码等）。相比采样分析器，探查分析器更可能会将性能偏差引入应用。比如，JVM 会内联小方法（参见第 4 章）使得执行时不会产生方法调用。编译器的这种判断是基于代码的大小，所以有可能使得这段代码不再被内联，这取决于如何探查代码。这会导致探查分析器高估某些方法对整体性能的影响。方法内联只是一个例子，说明编译器会基于代码在内存中的布局而做出决策。一般来讲，加入（更改）的探查代码越多，运行的性能分析结果就越有可能发生变更。

为何 ImmutableMap.get() 只在这里出现，而没在前面的采样分析结果中出现，还有一个很重要的技术原因。Java 的采样分析器只能在线程位于安全点时采集线程样本——基本上只有在 JVM 分配内存的时候。get() 方法可能永远都不会进入安全点，所以可能永远都不会被采样。

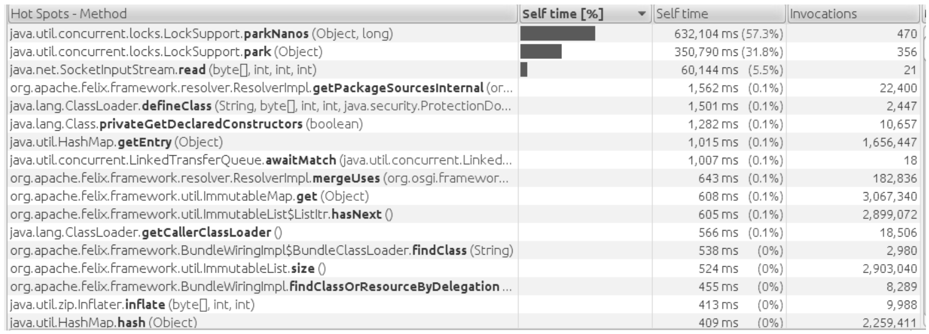
在这个例子中，探查分析器和采样分析器给出的结果都指向同样的代码区域：类装载和类解析。实际上，不同分析器的结果可能指向的是完全不同的代码区域。性能分析器可以用来很好地估计，但也仅仅是估计而已：某时某刻它们也会犯错。

快速小结

1. 探查分析器可以给出更多的应用信息，但相对采样分析器，它对应用的影响更大。
2. 探查分析器应该仅在小代码区域——一些类和包——中设置使用，以限制对应用性能的影响。

3.3.3　阻塞方法和线程时间线

图 3-4 是用另一种探查分析工具（NetBeans 性能分析器）分析的 GlassFish 启动。可以看出，此时的执行时间主要被 park() 和 parkNanos() 占用了（相对于占用更少的 read() 方法）。



Hot Spots - Method	Self time [%]	Self time	Invocations
java.util.concurrent.locks.LockSupport.**parkNanos** (Object, long)		632,104 ms (57.3%)	470
java.util.concurrent.locks.LockSupport.**park** (Object)		350,790 ms (31.8%)	356
java.net.SocketInputStream.**read** (byte[], int, int, int)		60,144 ms (5.5%)	21
org.apache.felix.framework.resolver.ResolverImpl.**getPackageSourcesInternal** (or...		1,562 ms (0.1%)	22,400
java.lang.ClassLoader.**defineClass** (String, byte[], int, int, java.security.ProtectionDo...		1,501 ms (0.1%)	2,447
java.lang.Class.**privateGetDeclaredConstructors** (boolean)		1,282 ms (0.1%)	10,657
java.util.HashMap.**getEntry** (Object)		1,015 ms (0.1%)	1,656,447
java.util.concurrent.LinkedTransferQueue.**awaitMatch** (java.util.concurrent.Linked...		1,007 ms (0.1%)	18
org.apache.felix.framework.resolver.ResolverImpl.**mergeUses** (org.osgi.framewor...		643 ms (0.1%)	182,836
org.apache.felix.framework.util.ImmutableMap.**get** (Object)		608 ms (0.1%)	3,067,340
org.apache.felix.framework.util.ImmutableList$ListItr.**hasNext** ()		605 ms (0.1%)	2,899,072
java.lang.ClassLoader.**getCallerClassLoader** ()		566 ms (0.1%)	18,506
org.apache.felix.framework.BundleWiringImpl$BundleClassLoader.**findClass** (String)		538 ms (0%)	2,980
org.apache.felix.framework.util.ImmutableList.**size** ()		524 ms (0%)	2,903,040
org.apache.felix.framework.BundleWiringImpl.**findClassOrResourceByDelegation** ...		455 ms (0%)	8,289
java.util.zip.Inflater.**inflate** (byte[], int, int)		413 ms (0%)	9,988
java.util.HashMap.**hash** (Object)		409 ms (0%)	2,259,411

图 3-4：一个带有阻塞方法的分析器

这些方法（以及类似的阻塞方法）并不消耗 CPU 时间，所以对应用的整体 CPU 使用率没有贡献。没有必要优化它们的执行。应用线程花费 623 秒不是在执行 `parkNanos()` 方法，而是等待别的事情发生（例如，等待其他线程调用 `notify()` 方法）。`park()` 和 `read()` 方法同样如此。

因为这个原因，大多数分析器不会报告被阻塞的方法，相应的线程也被显示为空闲（这个例子，NetBeans 被设置为显式包括这些方法和其他 Java 级别的方法）。在这个例子中，这是件好事：停止运行的是 Java 线程池中的线程，这些线程用来执行服务器所收到的 servlet（和其他）请求。启动时没有请求发生，所以这些线程被阻塞，等待任务执行。这是正常状态。

在其他情况下，你总是希望能看到那些阻塞调用所花费的时间。线程在 `wait()`——等待其他线程唤醒——中的用时决定了许多应用的整体执行时间。大多数基于 Java 的性能分析器可以通过设置过滤器和调整其他选项来显示或隐藏这些阻塞方法。

另一方面，审视线程的执行模式而不是分析器给阻塞方法所标记的用时，常常更有价值。图 3-5 是 Oracle Solaris Studio 性能分析工具中所显示的线程。

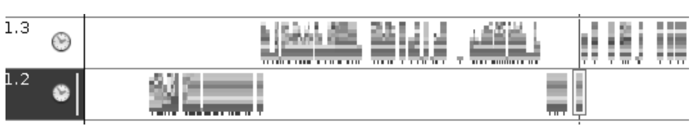


图 3-5 ：一个线程时间线分析器

每个横向区域都是一个线程（图中有两个线程：线程 1.3 和线程 1.2）。颜色（或不同的灰色）条表示不同方法的执行过程，空白区域表示线程没有在执行。从更高角度来观察，线程 1.2 先是执行了大量代码，然后等待线程 1.3，线程 1.3 之后稍稍等待线程 1.2 执行其他事情，等等。用工具进一步深入这些区域，可以让我们了解线程之间是如何相互影响的。

还可以注意到，有些空白区域没有任何线程在执行。这张图只显示了应用许多线程中的两个，所以这两个线程可能一起在等待其他线程，或者想成正在执行阻塞调用 `read()`（或类似的）。

快速小结

1. 线程被阻塞可能是性能问题，也可能不是，有必要进一步调查它们为何被阻塞。
2. 通过正被阻塞的方法调用，或者分析线程的时间线，可以辨认出被阻塞的线程。

3.3.4　本地分析器

本地性能分析工具是指分析 JVM 自身性能的工具。这类工具可以看到 JVM 内部的工作原理，如果应用自身含有本地库，这类工具也能看到本地库代码的内部。所有本地分析工具都可以用来分析 JVM 的 C 代码（以及任何本地库），而有些本地工具则可以用来分析任何 Java 和 C/C++ 应用。

图 3-6 有点眼熟，是 Oracle Solaris Studio 分析器分析 GlassFish 启动的结果。它是本地分析器，可以接受 Java 和 C/C++ 代码。（虽然名字中有 Solaris，但它也可以在 Linux 系统上运行。实际上，这些图片都是 Linux OS 上截取的分析结果。不过，如果在 Solaris 的内核结构上运行，这个工具可以展示更多的应用信息。）

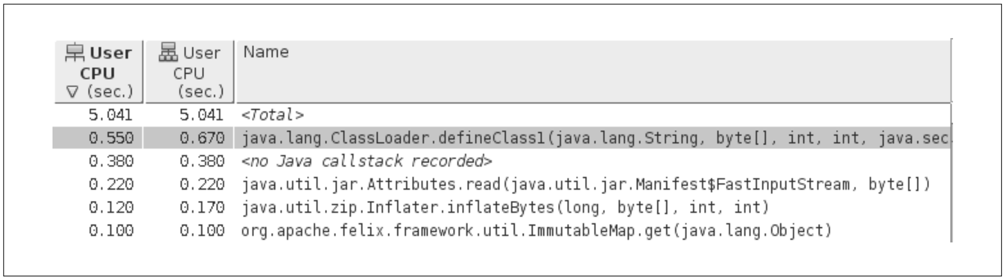
User CPU ▽ (sec.)	User CPU (sec.)	Name
25.105	25.105	<Total>
20.064	20.064	<JVM-System>
0.550	0.670	java.lang.ClassLoader.defineClass1(java.lang.String, byte[], int, int, java.sec
0.380	0.380	<no Java callstack recorded>
0.220	0.220	java.util.jar.Attributes.read(java.util.jar.Manifest$FastInputStream, byte[])
0.120	0.170	java.util.zip.Inflater.inflateBytes(long, byte[], int, int)
0.100	0.100	org.apache.felix.framework.util.ImmutableMap.get(java.lang.Object)

图 3-6：一个本地分析器

请注意，第一处与前面不同的是，记录下的应用总 CPU 时间为 25.1 秒，其中 20 秒是 `JVM-System` 所用。这是 JVM 自身代码的特性：JVM 编译器线程和 GC 线程（加上其他一些辅助线程）。我们可以进一步了解到，实际上，这个例子的所有时间都花费在了编译器上（启动过程确实如此，期间有许多代码需要编译）。这个例子中，GC 线程只花了很少量的时间。

除非是为了深入研究 JVM 自身，否则这些本地信息就足够了。如果我们愿意，可以进一步检查实际的 JVM 功能并优化它们。不过，这个工具所透露出来的关键信息是——基于 Java 的性能分析工具所不能提供的——应用花在 GC 上的时间。在基于 Java 的性能分析工具中，GC 线程的影响几乎看不到。（除非测试所运行的机器是 CPU 受限，否则编译器线程占用大量时间并无大碍：虽然编译线程会消耗大量的 CPU 时间，但只要机器上有更多可用的 CPU，应用自身就不受影响，因为编译是在后台发生。）

一旦检测出本地代码的影响，就可以过滤出来，以便重点考虑实际的启动过程（图 3-7）。



User CPU ▽ (sec.)	User CPU (sec.)	Name
5.041	5.041	<Total>
0.550	0.670	java.lang.ClassLoader.defineClass1(java.lang.String, byte[], int, int, java.sec
0.380	0.380	<no Java callstack recorded>
0.220	0.220	java.util.jar.Attributes.read(java.util.jar.Manifest$FastInputStream, byte[])
0.120	0.170	java.util.zip.Inflater.inflateBytes(long, byte[], int, int)
0.100	0.100	org.apache.felix.framework.util.ImmutableMap.get(java.lang.Object)

图 3-7：一个过滤的本地分析器

采样分析器再次把 `defineClass1()` 指为了最热的方法，虽然这个方法及其子调用的实际用时——5.041 秒中的 0.67 秒——占了约 11%（不像前面采样分析器报告的那么显著）。分析结果还指出了其他一些需要调查的信息：读取和解压 JAR 文件。由于和类装载有关，这说明我们追踪的方向是正确——但在这个例子中，重要的是看到了实际读取（通过 `inflateBytes()` 方法）JAR 文件 I/O 只占了极少的百分点。其他工具则不会告诉我们这些信息——部分是因为 Java ZIP 库中的本地代码被当作阻塞调用而被过滤掉了。

无论你用上述哪种性能分析工具，或更好的工具，至关重要的一点就是熟悉它们的特性。性能分析器是查找性能瓶颈最重要的工具，但你必须学会如何使用它们，然后找到需要优化的代码区域，而不是仅仅关注最上头的方法。

快速小结

1. 本地性能分析器可以提供 JVM 和应用代码内部的信息。
2. 如果本地性能分析器显示 GC 占用了主要的 CPU 使用时间，优化垃圾收集器就是正确的做法。然而，如果显示编译线程占用了明显的时间，则说明通常对应用性能没什么影响。

3.4 Java任务控制

商业版 Java 7（从 7u40 开始）和 Java 8 包含了称为 Java Mission Control（以下称 JMC）的监控新特性。JDK 6 JRockit JVM（JMC 的技术源自于此）的用户对这个特性应该很熟悉，Oracle 将它作为 Java 7 的一部分合并了进来。JMC 不是开源版 Java 的一部分，并且只有通过商业许可才可用（也就是说，和其他公司的有竞争力的监控工具的许可程序相类似）。

JMC 的程序（`jmc`）开启一个窗口以显示当前机器上的 JVM 进程，你可以选择一个或多个进行监控。图 3-8 是 JMC 监控 GlassFish 应用服务器的一个实例。

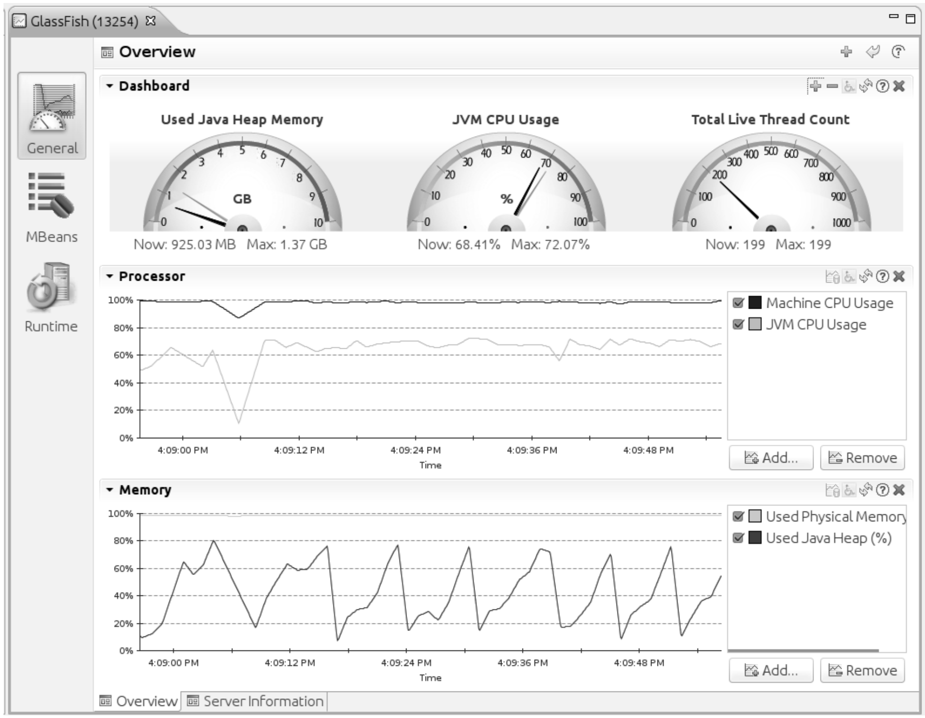


图 3-8：Java 任务控制器监控

图中显示了 JMC 正在监控的基本信息：CPU 使用率和堆使用量。但请注意，CPU 图包括了当前机器上所有的 CPU（基本上是 100%，即使这里被监控的应用只用了 70%）。这是监控的关键特性：JMC 可以监控整个系统，而不仅是所选择的 JVM。顶部的“面板”可以配置，以显示 JVM 的信息（所有的统计信息，包括 GC、类加载、线程使用情况、堆使用量等）以及操作系统相关的信息（机器总的 CPU 和内存使用量、页面交换、平均负载等）。

像其他监控工具一样，JMC 可以向任何被监控应用的 MBean 发起 Java Management Extensions（JMX）调用。

3.4.1　Java飞行记录器

JMC 的关键特性是 Java 飞行记录器（Java Flight Recorder，JFR）。正像它名字所暗示的，JFR 数据是 JVM 的历史事件，这些可以用来诊断 JVM 的历史性能和操作。

JFR 的基本操作是开启一组事件（例如，线程等待某个锁而被阻塞的事件）。每当选择的事件发生时，就会保存相应的数据（保存在内存或文件中）。数据流保存在循环缓冲中，所以只有最近的事件。JMC 可以显示这些事件——实时从 JVM 获取或者从文件读取——你可以对这些事件进行分析，诊断性能问题。

所有这些——事件的类别、循环缓冲的大小、数据保存在哪里等——都可以通过 JVM 的不同参数、JMC 的 GUI 以及程序运行时的 `jcmd` 命令来控制。JFR 的默认设置只有很低的开销：程序性能的 1% 以下。如果开启的事件变多，或者事件触发的阈值更改等，开销也会随之变化。本节的后面将详细讨论这些配置，我们先了解一下这些事件显示出来是什么样的，从而更容易理解 JFR 是如何工作的。

1. JFR概览

示例中的 JFR 记录了 GlassFish 应用服务器 6 分钟的数据。服务器运行的是第 2 章所讨论的股票 servlet。JMC 加载这些记录后，首先看到的是基本的监控概要（图 3-9）。

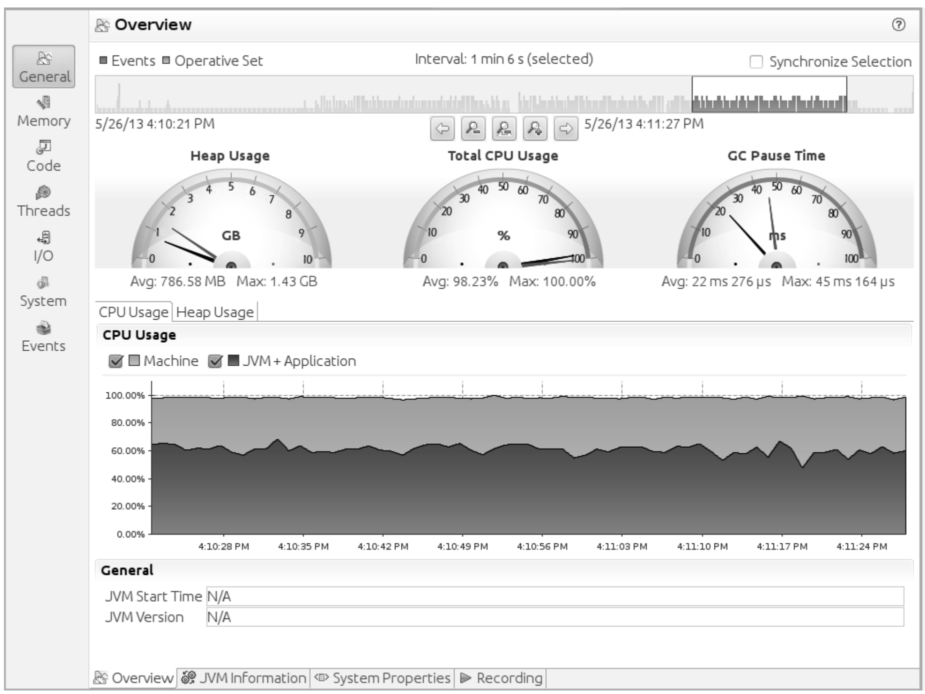


图 3-9：JFR 常规信息

上图所展示的与 JMC 基本监控时的显示非常类似。仪表盘显示的是 CPU 使用率和堆使用量，仪表盘上方是事件的时间线（用一系列的垂直条表示）。时间线可以放大以显示重点区域。本示例记录的数据区间为 6 分钟，放大之后，显示的是到记录结束之前 1:06 分钟内的数据。

图中的 CPU 占用率[2]大体可以看出正在做什么。GlashFish 的 JVM 在图的底部（平均大约 70% 的占用率），而机器的 CPU 为 100%。面板底部还有一些选项卡可以一探究竟：系统

注 2：中文 JMC 中的显示。——译者注

属性，JFR 记录的实际数据。窗口左边面板上的图标更为重要：通过这些图标对应的视图可以了解应用内部的行为。

2. JFR内存视图

这里汇集的信息范围很广。图 3-10 只显示了内存视图中的一个面板。

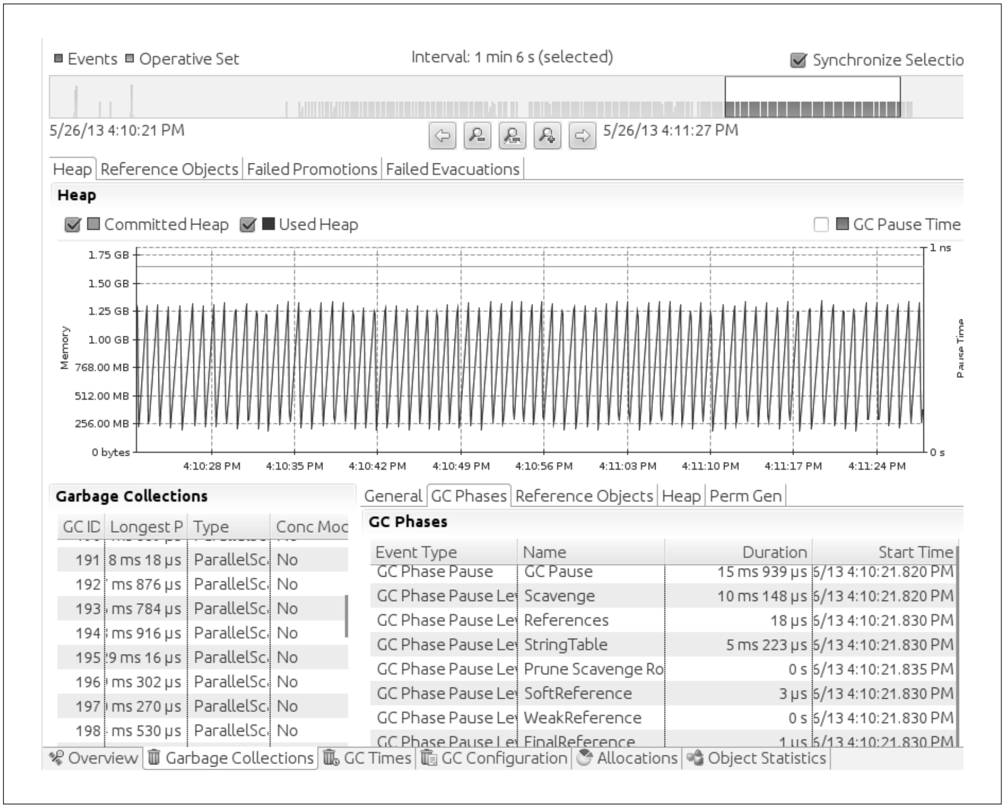


图 3-10：JFR 内存面板

图中显示，内存的使用量随着新生代的回收而发生很有规律的波动（而更重要的是，该应用的内存堆整体没有什么增长：没有对象被提升到老年代）。左下角面板显示 JFR 记录期间发生的所有垃圾收集（每行表示一次收集），包括持续的时间和收集的类型（这个例子始终是 `ParallelScavenge`）。当选中某一行事件时，右下角面板会进一步分拆显示事件的详细信息，包括这次垃圾收集的所有特定阶段和花了多长时间。

正如大家从这页的不同选项卡上所看到的，还有许多其他有价值的信息：有多少引用对象和多长时间被清理，并发收集过程中是否有对象提升或疏散失败（evacuation failure），GC 算法的配置参数（包括代的大小和 Survivor 空间的配置），甚至已分配的特定种类对象的信息。当你读到第 5 章和第 6 章时，希望你还能牢记这里所讨论的这个工具所能诊断的问题。如果你需要弄明白为何 CMS 出现并发失败而执行全部 GC（因为晋升失败？），或者 JVM 如何调整晋升阈值（tenuring threshold），或者关于 GC 及其行为的几乎所有数据，

JFR 都将回答。

3. JFR代码视图

JFR[3] 的"代码"视图显示所记录的基本性能分析信息（图 3-11）。

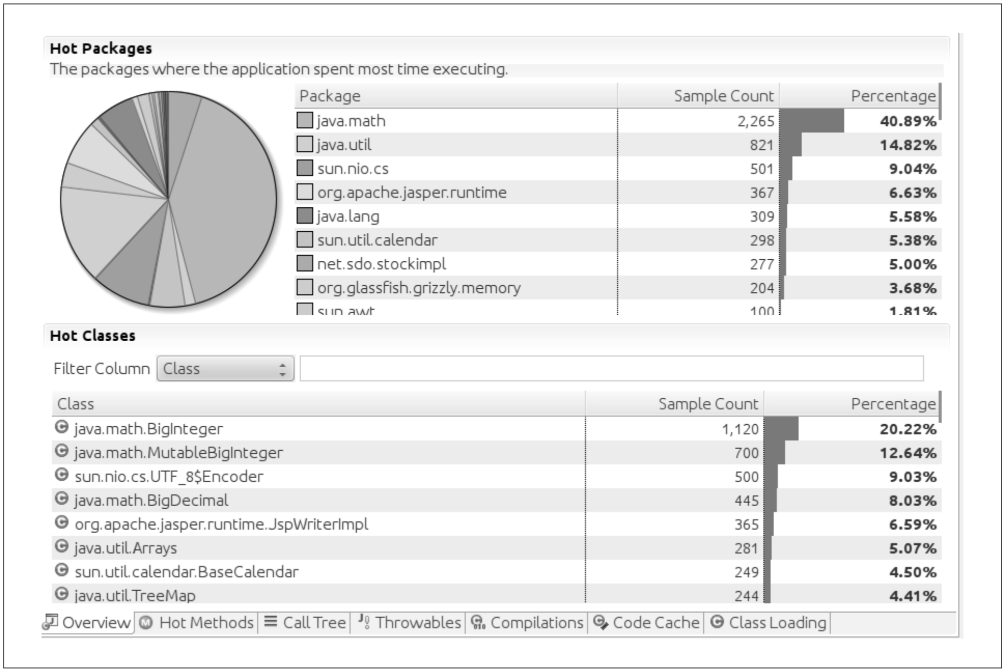


图 3-11：JFR 代码面板

这个视图的第一个选项卡显示了一组包名，这是一个许多分析器中都没有的重要特性。不出所料，股票应用的 41% 的时间用在 `java.math` 包内的计算上。视图底部还有其他传统分析视图的选项卡：最热的方法和被分析代码的调用树。

与其他分析器不同的是，JFR 提供了查看代码内部的其他模式。"异常错误"（Throwable）选项卡提供了应用内部异常处理的视图（第 12 章将讨论为什么异常处理过多会导致糟糕的性能）。还有些选项卡告诉我们编译器正在做什么，包括代码缓存的内部视图（参见第 4 章）。

还有其他显示——如线程、I/O 和系统事件——但这其中的大部分只是为 JFR 记录的真实事件提供了更好的视图。

4. JFR事件概览

JFR 生成的事件流会作为记录保存下来。上述视图可以展示这些事件，但查看事件最重要的方法是"事件"面板自身，参见图 3-12。

注 3：原文是"Java Mission Control"，说 JFR 更为准确。——译者注

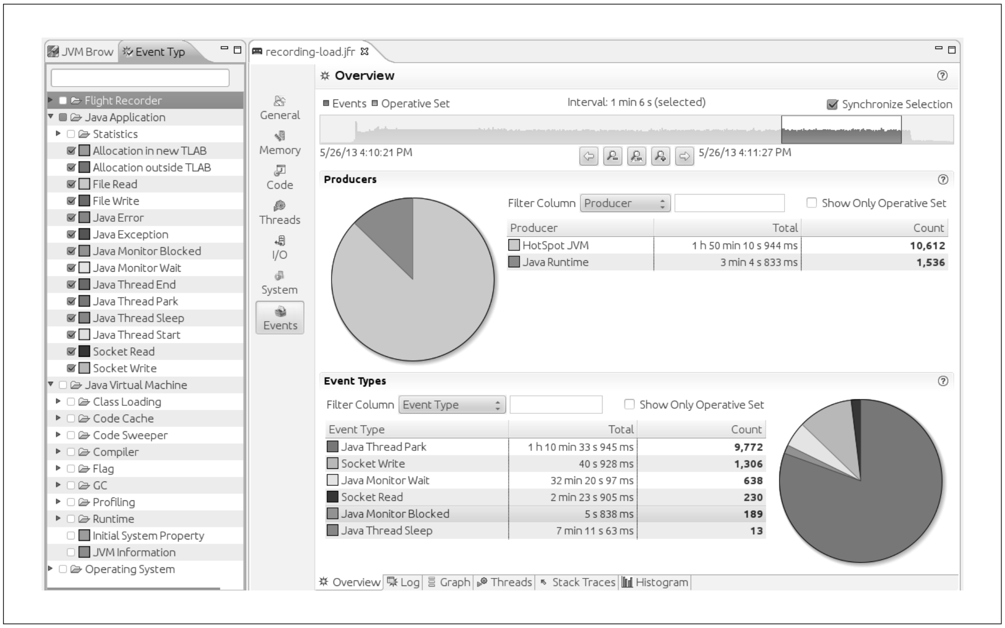


图 3-12：JFR 事件面板

窗口左边的面板可以过滤需要显示的事件。这里只选择应用级别的事件。请注意，在开始记录时，只包括特定种类的事件：在这个时候，我们进行事后的过滤（下节演示如何在记录时过滤所包括的事件）。

本示例的 66 秒间隔中，应用的 JVM 产生了 10 612 个事件，JDK 库产生了 1536 个事件，靠近窗口底部的是在这个期间生成的 6 种事件类型。我已经讨论过这个示例的线程停止（thread-park）和管程等待（monitor-wait）事件为什么这么高。可以忽略这些事件（事实上，通常它们是很好的过滤条件）。那其他事件呢？

过了 66 秒后，应用的多个线程写 socket 花费了 40 秒。对于在 4 个 CPU 上运行的应用服务器来说，这并不是一个不合理的数字（也就是说，264 秒），但意味着可以通过向客户端写入较少的数据来改善性能（使用第 10 章列出的技术）。

与此类似，多个线程读 socket 用了 143 秒。这个数字听起来比较糟糕，值得进一步追踪这些事件，不过最终证实有些线程使用了阻塞 I/O 读取预计周期性到达的管理请求。这些请求之间——很长一段时间——线程被 `read()` 方法阻塞。所以这里的读取时间变得可以接受：如果使用性能分析器，你应该判断大量的线程被 I/O 阻塞是预料之中的，还是说明存在性能问题。

还有监视器阻塞事件。正如第 9 章所讨论的，锁竞争分为两种级别：首先是线程自旋等待锁，然后线程使用（进程中称为锁膨胀）一些 CPU 或 OS 特定的代码等待锁。标准的分析器可以提示处于那种情形，因为自旋时间包括在该方法的 CPU 时间内。本地分析器可以给出锁膨胀的信息，但这是不确定的（例如，Oracle Studio 分析器在 Solaris 上运行的很

好，但 Linux 上缺少必要的能提供与 Solaris 相同信息的操作系统钩子）。但 JVM 可以直接向 JFR 提供所有的这些数据。

使用锁可见性的例子请参见第 9 章。不过通常来说，理所当然的那些 JFR 事件是直接来自于 JVM 的事件，它们给应用增加了一个可见性级别，这是其他工具不能提供的。在 Java 7u40 中，JFR 可以监控 77 种事件类型。根据发布的版本，事件种类的精确数字会略微有所不同，这里是一些最有用的。

以下每种事件类型的说明都包括两点。事件可以像其他工具如 `jconsole` 和 `jcmd` 那样收集基本信息。这类信息在第一点中描述。第二点则描述的是事件提供者，这在 JFR 之外很难获得。

Classloading

- 装载或卸载的类的数量
- 哪个类装载器装载的类，装载单个类需要的时间

Thread statistics

- 创建和销毁的线程数，线程转储
- 被锁阻塞的线程（以及阻塞住它们的特定锁）

Throwables

- 应用所用的异常错误类
- 有多少异常和错误被抛出了，以及它们生成时的栈追踪信息

TLAB allocation

- 堆分配的数量和本地线程分配缓冲的大小
- 堆中分配的对象和分配它们的栈追踪信息

File and socket I/O

- 执行 I/O 所用的时间
- 每次读 / 写调用所用的时间，读或写很长时间的特定文件或 socket

Monitor blocked

- 等待管程的线程
- 在特定管程上阻塞的线程以及它们被阻塞的时间

Code cache

- 代码缓存的大小以及包含多少量
- 从代码缓存中移走的 Java 方法，代码缓存配置

Code compilation

- 哪个方法已被编译，OSR 编译和编译的时长
- 除了 JFR 其他工具也包括的信息，但这是从多个源头获取的统一信息

Garbage collection

- GC 的时间，包括单个阶段，代的大小
- 除了 JFR 其他工具也包括的信息，但这是从多个工具获取的统一信息

Profiling

- 探查和采样分析器

- 不像真的性能分析器那样有很多信息，但 JFR 分析器提供了很好的更高层次的概要信息

3.4.2　开启JFR

在 Oracle JVM 的商业版本中，JFR 初始时为关闭。为了开启它，可以在应用的启动命令行上添加标志 `-XX:+UnlockCommercialFeatures -XX:+flightRecorder`。这会开启 JFR 特性，但直到记录过程自身开始时才会记录信息。记录可以通过 GUI 或者命令行产生。

1. 通过JMC开启JFR

开启记录的最简单方法是通过 JMC GUI（`jmc`）。当 `jmc` 启动时，它会列出运行在当前系统中的所有 JVM 进程。JVM 进程显示在树状的节点配置中。双击[4]“飞行记录器”（Flight Recorder）标签下的节点会弹出像图 3-13 这样的飞行记录窗口。

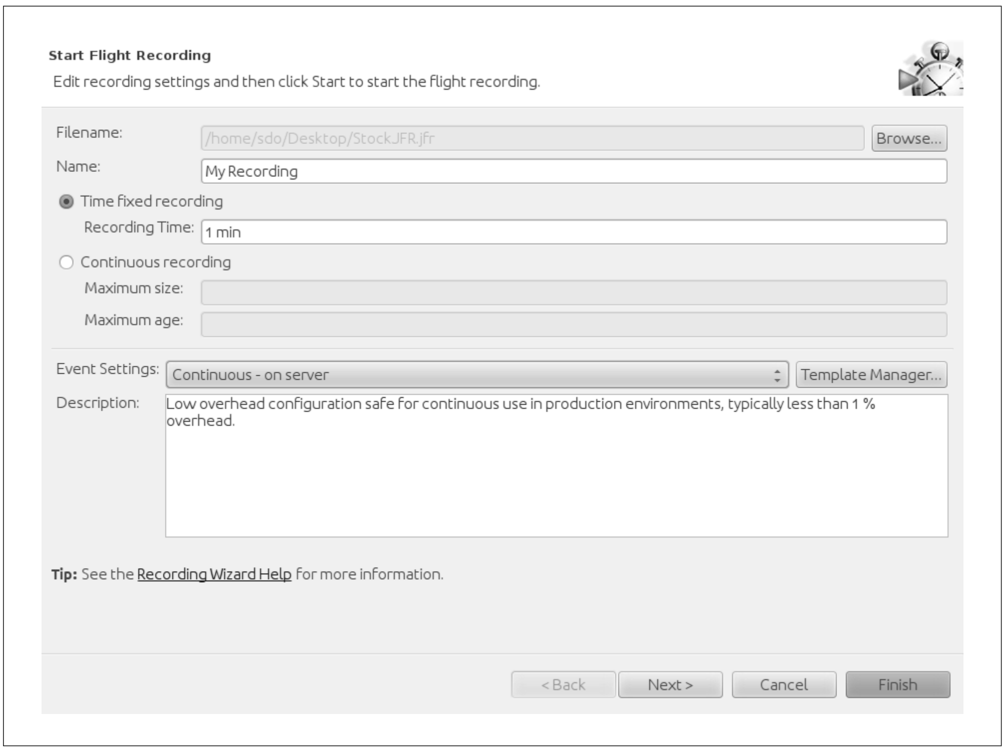


图 3-13：JFR 启动飞行记录窗口

飞行记录器可以设为两种模式之一：固定持续时间（这个例子是 1 分钟），或者持续进行。对于持续记录，会有一个循环缓冲，缓冲包含最近的事件，并且这些事件在设置的持续时间和大小内。

注 4：原文为“expand”，参考实际操作应为双击。——译者注

为了进行主动分析——意味着你将开启记录，然后还会产生一些工作量，或者在 JVM 热身之后——应该用固定持续时间的方式记录，在负载测试试验中开启记录。这些记录将对测试期间 JVM 如何响应给出很好的指示。

持续记录最适合于响应性分析（reactive analysis）。JVM 会保留最近的事件，然后转储为响应某些事件的记录。例如，WebLogic 应用服务器可以在遇到应用服务器异常事件时（例如处理超过 5 分钟的请求），触发转储记录。你可以针对任何事件，设立自己的监控工具转储这些记录。

2. 通过命令行开启JFR

开启 JFR 之后（选项 -XX:+flightRecorder），有两种方法控制记录应该在何时以及如何发生。JVM 用 -XX:+flightRecorderOptions=*string* 参数方式启动时，可以控制这些记录参数。这对响应性记录来说最为有用。参数中的 *string* 是一列逗号分隔的名字 – 值对，可以用以下选项：

name=*name*

　用以标识记录的名字。

defaultrecording=<*true*|*false*>

　表示初始时是否开启记录。默认为 false。对于响应性分析，应该设为 true。

settings=*path*

　JFR 设置文件的文件名（参见下节）。

delay=*time*

　记录开始前延迟的时间量（例如，30 秒，1 小时）。

duration=*time*

　记录持续的时间。

filename=*path*

　记录文件名。

compress=<*true*|*false*>

　记录是否开启压缩（gzip）。默认为 false。

maxage=*time*

　循环缓冲中保留记录数据的最长时间。

maxsize=*size*

　记录循环缓冲的最大尺寸（例如，1023 K，1 M）。

在某些情况下，像上述那样设置默认记录很有用，但为了更大的灵活性，所有选项可在程序运行时（假设 -XX:+flightRecorder 已先指定），用 jcmd 来控制。

要开启飞行记录：

```
% jcmd process_id JFR.start [options_list]
```

`options_list` 是一组用逗号分隔的名字 – 值对，控制记录如何进行。可能的选项与命令行使用 `-XX:+flightRecorderOptions=`*`string`* 时的标志完全一样。

如果开启持续记录，可以在任何时间，通过以下命令将当前循环缓冲里的数据转储到文件中：

```
% jcmd process_id JFR.dump [options_list]
```

选项列表包括：

`name=`*`name`*

在这个名字下的记录已经开始。

`recording=`*`n`*

JFR 记录的编号。

`filename=`*`path`*

转储文件的位置。

对于给定的进程，可能开启了多个 JFR 记录。以下命令可查看开启的记录：

```
% jcmd process_id JFR.check [verbose]
```

这个例子中，用记录启动时的名字来标识它们，也可以任意指定记录的编号（可以在其他 JFR 命令中使用）。 最后是进程放弃记录的命令：

```
% jcmd process_id JFR.stop [options_list]
```

这个命令接受以下选项：

`name=`*`name`*

停止记录的名字。

`recording=`*`n`*

停止记录的编号（可由 `JFR.check` 获得）。

`discard=`*`boolean`*

如果为 `true`，则丢弃数据而不是写到前面所提供的文件中（如果有的话）。

`filename=`*`path`*

数据写到给定的路径上。

回归检查运行系统的时候，在自动化性能测试系统中运行这些命令并生成记录，会很有用。

3.4.3　选择JFR事件

JFR（当前的）支持大约 77 种事件。大多数是周期性事件：它们的周期以毫秒记（例如，基于采样的性能分析事件）。其他事件仅当事件的持续时间超出阈值时才会触发（例如，读文件事件仅在 `read()` 超过指定时间时才会触发）。

> ### 其他 JFR 事件
>
> JFR 是可扩展的：应用可以定义自己的事件。因此，依据所考虑的应用，你的 JFR 视线可以显示更多类型的事件。例如，WebLogic 应用服务器开启一组应用服务器事件：JDBC 操作、HTTP 操作等。这些事件和前面所讨论的 JFR 事件一样：它们可以单个开启，可以有关联的阈值，等等。同样，最新版本的 JVM 可能会添加一些这里没有讨论到的事件。
>
> 详情请查阅最新的产品文档。

收集事件自然会引入一些开销。触发事件收集的阈值——因为增加了事件——在开启 JFR 记录时也会增加开销。默认记录中不会收集所有的事件（6 个最耗资源的事件没有开启），基于事件的阈值有些高。这使得默认记录的开销小于 1%。

有时额外的开销是值得的。例如，考虑 TLAB 事件，它可以帮助你判断对象是否正直接分配到老年代，但默认情况下这些事件没有开启记录。同样，默认记录开启了性能分析事件，但只有每 20 毫秒——可以给出适当的概览，但也可能导致采样错误。

JFR 捕获的事件（包括事件的阈值）都定义在模板中（可通过命令行的设置选项选择）。JFR 自带了两个模板：默认模板（限制了事件使得开销效率 1%）和性能分析模板（大多数基于阈值的事件被设置为每 10 毫秒触发）。这个性能分析模板的开销估计是 2%（而你的数据通常也都不一样，我观察到的典型开销要比这小得多）。

模板由 `jmc` 模板管理器管理。你可能已经注意到图 3-13 中启动模板管理器的按钮了。模板保存在两个地方：$HOME/.jmc/<release> 目录（用户本地）下，和 $JAVA_HOME/jre/lib/jfr（JVM 全局）下。模板管理器允许你选择全局模板（可以说模板“在服务器上”），选择本地模板，或定义新模板。定义模板时，来回多看看可用的事件，依据需要开启（或关闭）它们，还可以选择根据需要来设置事件阈值。

图 3-14 显示 File Read 事件开启，并且阈值为 15 毫秒：读取文件超过该值的就会触发事件。这个事件也可以配置为给 File Read 事件产生栈追踪信息。这增加了开销——这就是为何抓取事件栈追踪信息是可配置的选项。

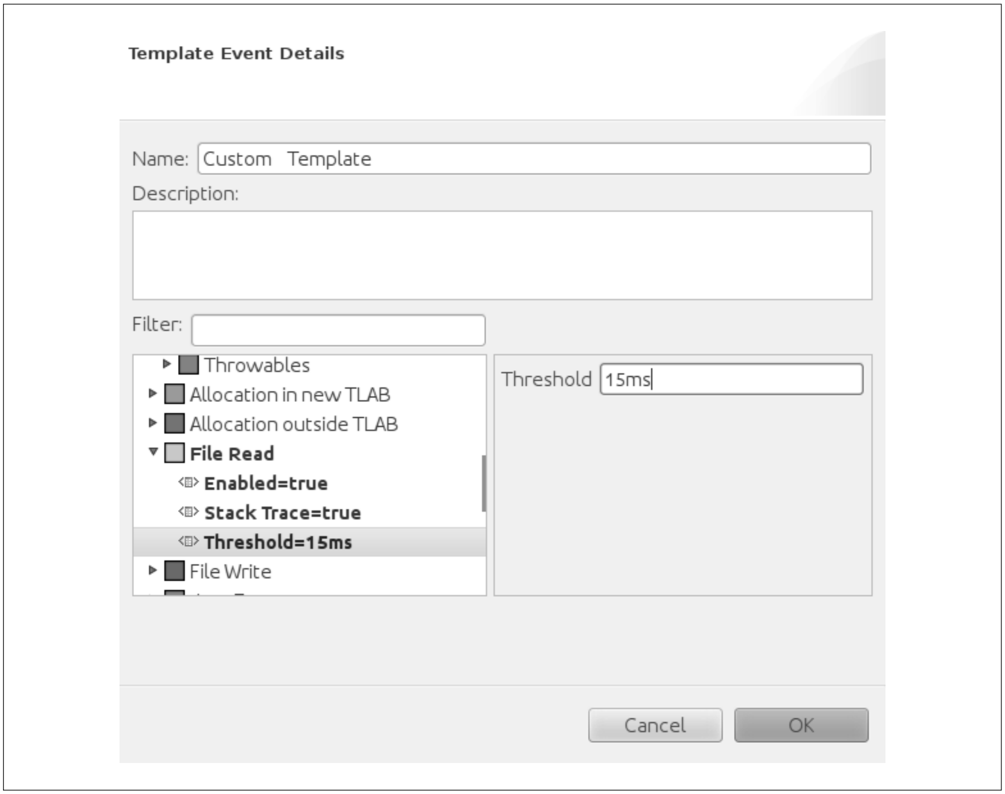


图 3-14：JFR 事件模板的一个样本

事件模板是简单的 XML 文件，所以判断模板中哪个事件开启（以及阈值和栈追踪信息配置）的最好办法是阅读 XML 文件。XML 作为配置文件，使得本地模板可以在一台机器上定义，然后复制到全局模板目录下，供团队中的其他人使用。

快速小结

1. 由于 JFR 内建于 JVM，所以可以最大可能地查看 JVM 内部。
2. 像其他工具一样，JFR 给应用引入了一些开销。对于日常使用，可以开启 JFR，以较低的开销收集大量信息。
3. JFR 用于性能分析，但它在生产系统中也很有用，所以你可以检查那些导致失败的事件。

3.5 小结

好的工具是性能分析成功的关键。对于工具所能告诉我们的信息，本章只是皮毛。关键是要记住以下几点。

- 没有完美的工具，每种工具也各有所长。分析器 X 可能适合很多应用，但可能有些情况下就会缺少一些信息，而分析器 Y 就能很清楚地给出这些信息。你总是需要灵活使用。
- 命令行监控工具可以自动收集重要数据，确保在自动化性能测试中能够收集监控数据。
- 工具的演变非常快：本章提到的一些工具可能已经过时了（或者已经被新的更强大的工具所取代）。这个领域很重要的一点就是要与时俱进。

第 4 章

JIT编译器

即时（Just-In-Time，JIT）编译器是 Java 虚拟机的核心。对 JVM 性能影响最大的莫过于编译器，而选择编译器是运行 Java 程序时首先要做的选择之一——无论你是 Java 开发人员还是最终用户。幸运的是，在绝大多数情况下，只需要对编译器做一些基本的调优。

本章将深入介绍编译器。首先介绍编译器的工作原理，并讨论 JIT 编译器的优点和不足。然后转而介绍每个 Java 版本所包括的编译器：为特定场景选择合适的编译器，是你为了程序快速运行所采取的最重要的步骤，你需要明白这个道理。本章的最后涵盖了一些编译器调优的中高级技巧。这些调优技巧有助于使应用的性能百尺竿头更进一步。

4.1 JIT编译器：概览

先作一些介绍，如果你基本理解即时编译的话，可以放心大胆地跳过开头这段。

计算机——更具体说是 CPU——只能执行相对少而特定的指令，这被称为汇编码或者二进制码。因此，CPU 所执行的所有程序都必须翻译成这种指令。

像 C++ 和 Fortran 这样的语言被称为编译型语言，因为它们的程序都以二进制（编译后的）形式交付：先写程序，然后用编译器静态生成二进制文件。这个二进制文件中的汇编码是针对特定 CPU 的。只要是兼容的 CPU，都可以执行相同的二进制代码：比如，AMD 和 Intel CPU 共享一个基本的、常用的汇编语言指令集，新版本的 CPU 几乎总是能执行与老版本 CPU 相同的指令集。但反过来并不总是成立，新版本的 CPU 时常会引入一些指令，这些指令无法在老版本 CPU 上运行。

另外还有一些像 PHP 和 Perl 这样的语言，则是解释型的。只要机器上有合适的解释器（即称为 `php` 或 `perl` 的程序），相同的程序代码可以在任何 CPU 上运行。执行程序时，解释器会将相应代码转换成二进制代码。

每种语言类型都各有长处和不足。解释性型语言的程序可移植：相同的代码你丢到任何有适当解释器的机器上，它都能运行。但是，它运行起来可能就慢了。举个简单的例子，不妨考虑一下执行循环时会发生什么：当解释器执行循环体时，会重新翻译每一行代码。编译过后的代码就不必再重复做这样的转换了。

好的编译器在生成二进制代码时需要考虑许多因素。一个简单例子是二进制代码中的语句顺序：生成的汇编语言指令与执行时的顺序并不完全相同。执行两个寄存器值相加的语句可能只需要一个时钟周期，但（从主存储器）获取加法所需要的数据可能需要好几个周期。

因此，好的编译器生成的二进制代码需要包括装载数据、执行其他指令，然后——当数据准备好时——执行加法。而一次只能看一行的解释器就没有足够的信息生成这样的代码了。它会请求内存数据，然后一直等到数据准备好之后再执行加法。稍差点的编译器也这么干，而且顺便说一句，即便是最好的编译器偶尔也需要等待指令完成。

由于这些（或其他的）原因，解释型代码几乎总是明显比编译型代码要慢：编译器有足够的程序信息，这些信息可用来大量优化二进制代码，这些是简单解释器无法做到的。

解释型代码的优势在于可移植。很显然，SPARC CPU 的二进制编译器无法在 Intel CPU 上运行。而用 Intel Sandy Bridge 处理器最新 AVX 指令的二进制代码也无法在老的 Intel 处理器上运行。因此，商业软件通常会在较老的处理器上编译，从而无法利用最新的指令。这里面有很多技巧，例如，发布二进制代码时附带多个共享库，而这些共享库执行的代码都是对性能较敏感的，还要有多种版本与各种类型的 CPU 相匹配。

Java 试图走一条中间路线。Java 应用会被编译——但不是编译成特定 CPU 所专用的二进制代码，而是被编译成一种理想化的汇编语言。然后该汇编语言（称为 Java 字节码）可以用 `java` 运行（与 `php` 解释运行 PHP 脚本是相同的道理）。这使得 Java 成为一门平台独立的解释型语言。因为 `java` 程序运行的是理想化的二进制代码，所以它能在代码执行时将其编译成平台特定的二进制代码。由于这个编译是在程序执行时进行的，因此被称为“即时编译”（即 JIT）。

Java 虚拟机在执行时编译代码的这种方式是本章关注的重点。

热点编译

如第 1 章讨论的那样，本书中的 Java 实现是 Oracle 的 HotSpot JVM。HotSpot 的名字来自于它看待代码编译的方式。对于程序来说，通常只有一部分代码被经常执行，而应用的性能就取决于这些代码执行得有多快。这些关键代码段被称为应用的热点，代码执行得越多就被认为是越热。

因此 JVM 执行代码时，并不会立即编译代码。有两个基本理由。第一，如果代码只执行一次，那编译完全就是浪费精力。对于只执行一次的代码，解释执行 Java 字节码比先编译然后执行的速度快。

但如果代码是经常被调用的方法，或者是运行很多次迭代的循环，编译就值得了：编译的代码更快，多次执行累积节约的时间超过了编译所花费的时间。这种权衡是编译器先解释

执行代码的原因之一——编译器可以找出哪个方法被调用得足够频繁，可以进行编译。

第二个理由是为了优化：JVM 执行特定方法或者循环的次数越多，它就会越了解这段代码。这使得 JVM 可以在编译代码时进行大量优化。

本章后面将讨论这些大量的优化（以及影响它们的方法），先考虑一个简单的例子，即 `equals()` 方法。这个方法存在于每个 Java 对象中（既然所有类都继承自 `Object` 类），并且经常被子类重写。当解释器遇到 `b = obj1.equals(obj2)` 语句时，为了知道该执行哪个 `equals()`，必须先查找 `obj1` 的类型（类）。这个动态查找的过程有点消耗时间。

寄存器和主内存

编译器最重要的优化包括何时使用主内存中的值，以及何时在寄存器中存贮值。考虑以下代码：

```
public class RegisterTest {
    private int sum;

    public void calculateSum(int n) {
        for (int i = 0; i < n; i++) {
            sum += i;
        }
    }
}
```

在某个时刻，实例变量 `sum` 必须驻留在主内存中，但从主内存获取数据是昂贵的操作，需要花费多个时钟周期才能完成。如果每次循环迭代都从主内存获取（或保存）`sum` 的值，性能就比较糟糕了。编译器不会这么做，它会将 `sum` 的初始值装入寄存器，用寄存器中的值执行循环，然后（在某个不确定的时刻）将最终的结果从寄存器写回到主内存。

这种优化非常高效，但这意味着线程同步的语义（参见第 9 章）对应用行为非常重要。一个线程无法看到另一个线程所用寄存器中保存变量的值，同步机制使得从寄存器写回主内存时其他线程可以准确地读到这个值。

使用寄存器是编译器普遍采用的优化方法，当开启逃逸分析（escape analysis）时（参见本章末尾），寄存器的使用更为频繁。

比如说，随着时间的流逝，JVM 发现每次执行这条语句时，`obj1` 的类型都是 `java.lang.String`。于是 JVM 就可以生成直接调用 `String.equals()` 的编译代码。现在代码更快了，不仅是因为被编译，也是因为跳过了查找该调用哪个方法的步骤。

不过没那么简单。下次执行代码时，`obj1` 完全有可能是别的类型而不是 `String`，所以 JVM 必须生成编译代码处理这种可能。尽管如此，由于跳过了方法查找的步骤，这里的编译代码整体性能仍然要快（至少和 `obj1` 一直是 `String` 时同样快）。这种优化只有在代码运行过一段时间观察它如何做之后才能使用：这是为何 JIT 编译器等待代码编译的第二个原因。

快速小结

1. Java 的设计结合了脚本语言的平台独立性和编译型语言的本地性能。
2. Java 文件被编译成中间语言（Java 字节码），然后在运行时被 JVM 进一步编译成汇编语言。
3. 字节码编译成汇编语言的过程中有大量的优化，极大地改善了性能。

4.2 调优入门：选择编译器类型（Client、Server或二者同用）

有两种“口味”的 JIT 编译器，选择哪种常常是应用运行时所需做的仅有的编译器调优。事实上，甚至在安装 Java 之前就必须考虑如何选择编译器，因为不同的 Java 安装包包含了不同的编译器。我们来逐步分析。首先找出何种环境下该用哪种编译器。

这两种编译器被称为 `client` 和 `server`。名字来自于命令行上用于选择编译器的参数（例如 `-client` 或 `-server`）。JVM 开发者（甚至一些工具）通常称这些编译器为 `C1`（编译器 1，client 编译器）和 `C2`（编译器 2，server 编译器）。它们的名字似乎意味着如何选择编译器受程序运行的硬件平台的影响，但这并不完全正确：特别是到现在，这些术语已经超过了整整 15 年，你的“client”笔记本也已经是 4 到 8 核的 CPU 和 8 GB 内存（处理能力比 Java 刚诞生时的中型服务器还要强）。

与众不同的编译器标志

选择编译器的 Java 标志与大多数标志不同，大都没有使用 `-XX`。标准的编译器标志是这几个简单的词语：`-client`、`-server` 或 `-d64`。

分层编译（tiered compilation）是个例外，标志采用常见的开启形式 `-XX:+TieredCompilation`。分层编译意味着必须使用 server 编译器。下面的命令行隐含着关闭分层编译，因为与 client 编译器的选择冲突。

```
% java -client -XX:+TieredCompilation other_args
```

两种编译器的最主要的差别在于编译代码的时机不同。client 编译器开启编译比 server 编译器要早。意味着在代码执行的开始阶段，client 编译器比 server 编译器要快，因为它的编译代码相比 server 编译器而言要多。

此处工程上考虑的权衡是，server 编译器等待编译的时候是否还能做更有价值的事：server 编译器在编译代码时可以更好地进行优化。最终，server 编译器生成的代码要比 client 编译器快。从用户角度看，权衡的取舍在于程序要运行多久，程序的启动时间有多重要。

此处最明显的问题是，为什么需要人来做这种选择？为什么 JVM 不能在启动时用 client 编译器，然后随着代码变热使用 server 编译器？这种技术被称为*分层编译*。代码先由 client 编译器编译，随着代码变热，由 server 编译器重新编译。

Java 7 的早期发布版中曾经提供了分层编译的实验性版本，结果却发现技术上存在许多难

点（尤其是两种编译器架构的不同），所以这些实验版的分层编译效果并不好。从 Java 7u4 开始，这些难点很大程度上得到了解决，所以分层编译常常可以让应用发挥最佳性能。

Java 7 中的分层编译有些瑕疵，所以没能成为默认设置。尤其是 Java 7 的分层编译容易超出 JVM 代码缓存的大小，使得代码无法优化编译（虽然很容易处理，参见 4.4 节“编译器的中间代码优化”的讨论）。使用分层编译需要指定 server 编译器（`-server` 或者确认它是特定 Java 安装包所用的默认值），并确保 Java 命令行包括标志 `-XX:+TieredCompilation`（默认值为 `false`）。Java 8 中，分层编译默认为开启。

为了解其中的权衡，我们来看一些例子。

4.2.1 优化启动

当快速启动时间是首要目标时，最常使用 client 编译器。不同应用使用不同编译器标志的差别见表 4-1。

表4-1：不同应用在不同编译器标志下的启动时间

应用	`-client`	`-server`	`-XX:+TieredCompilation`
HelloWorld	0.08	0.08	0.08
NetBeans	2.83	3.92	3.07
BigApp	51.5	54.0	52.0

对于简单的 `HelloWorld` 应用，没有编译器占据优势，因为都没有足够的代码可以运行以便优化。并且对于执行时间只有 80 毫秒的任务，我们也很难注意到性能差异是否真的存在。

NetBeans 是相当典型、规模适中的 Java GUI 应用。它启动时，装载约 10 000 个类，初始化一些图形对象等。启动时，client 编译器有很明显的优势：显而易见，server 编译器的启动慢了 38.5%，大约相差了 1 秒。注意分层编译器并不是很快，虽然只慢了约 8%，相差并不大。

这就是为什么 NetBeans——许多 GUI 程序喜欢它，包括 Web 浏览器所用的 Java 插件——默认使用 client 编译器的原因。性能是王道：其他部分还不错的情况下，启动若快，便是晴天，用户就会倾向于认为整个程序都像启动那样快。

启动时间要紧吗？

关于 GUI 程序有个重要的观点，即整体性能比启动性能更重要，而且这种场景更适合使用 server 编译器。

如果 server 编译器优化了应用中的 GUI 代码，最终 GUI 的响应性会有所提高，不过最终用户可能不太会注意到这种差别。但是，如果程序执行了大量其他计算，这样做就有意义了。比如，NetBeans 可以进行大范围的（和昂贵的）代码重构，如果使用 server 编译器，速度就会很快。

通常程序供应商会考虑默认的编译器应该是哪种（因为启动时间是大家首先讨论的事情之一，所以这些程序通常被优化成有最佳的启动时间）。如果你的应用与此不同，不要犹豫，应该尽量使用 server 编译器或分层编译器。

最后来看 BigApp：一个很大的服务器程序，装载超过 20 000 个类，初始化范围很广。因为是服务器应用，可以肯定它需要使用 server 编译器。尽管运行着许多进程，对 client 编译器仍然有些许优势。这个例子的重点是第 1 章中所提到的：问题并不总在 JVM。在这个示例中，需要从磁盘读取大量的 JAR 文件，从而制约了性能（除此以外，启动上的差异更有利于 client 编译器）。

快速小结

1. 如果应用的启动时间是首要的性能考量，那 client 编译器就是最有用的。
2. 分层编译的启动时间可以非常接近于 client 编译器所获得的启动时间。

4.2.2 优化批处理

对于批处理应用来说——处理的工作量固定——编译器的选择，归根到底取决于哪种编译器使得应用运行的时间最优。表 4-2 是一个例子。

表4-2：批处理应用在不同编译器下的执行时间

股票数量	-client（秒）	-server（秒）	-XX:+TieredCompilation（秒）
1	0.142	0.176	0.165
10	0.211	0.348	0.226
100	0.454	0.674	0.472
1000	2.556	2.158	1.910
10 000	23.78	14.03	13.56

使用第 2 章中的代码即股票的例子，此处获取 1 年的历史信息（以及平均数和标准差），股票数量从 1 到 10 000。

股票数量为 1 到 100 时，client 编译器最快完成启动任务，说明如果处理的股票数量少于 100 只，client 编译器是最佳选择。之后，性能优势就偏向了 server 编译器（特别是分层编译的 server 编译器）。即便是少量股票，分层编译的性能也很接近于 client 编译器，所以分层编译是适合所有情况的很好的备选方案。

还有一点比较重要，分层编译总是比标准的 server 编译器好一些。理论上，一旦程序足够运行，编译了所有的热点，server 编译器就应该达到最佳（或至少等同于）的性能。但任何程序都有一小部分代码很少执行。最好是编译这些代码——即便编译不是最好的方法——而不是以解释模式运行。正如本章后面（参见 4.4.2 节“编译阈值”）所讨论的，实际上即便应用永远运行，server 编译器也不可能编译它的所有代码。

快速小结

1. 对于计算量固定的任务来说，应该选择实际执行任务最快的编译器。
2. 分层编译是批处理任务合理的默认选择。

4.2.3 优化长时间运行的应用

最后我们来看看在不同的编译器下，长时间运行的应用之间的性能差别。衡量长时间运行的应用的性能，通常来说，是在应用“热身”之后——意味着它已经运行了足够长的时间，重要的代码都已经被编译——测量它处理的吞吐量。

例子使用基本的股票计算，并在 servlet 中进行。每次调用 servlet 就会随机抽取一只股票 25 年的信息。表 4-3 是用第 2 章介绍的 `fhb` 程序获取的数据，显示热身期为 0、60 和 300 秒时，每秒的调用次数。

表4-3：服务器应用在不同热身期下的吞吐量

热身期（秒）	`-client`	`-server`	`-XX:+TieredCompilation`
0	15.87	23.72	24.23
60	16.00	23.73	24.26
300	16.85	24.42	24.43

由于测试的周期为 60 秒，所以即便没有热身期，编译器仍然可以获得足够的信息编译热点，因此 server 编译器在本例中总是比较好。（另外，大量代码会在应用服务器启动时进行编译。）如之前所讨论的，相比单独的 server 编译器，分层编译可以编译更多代码，提供更多性能。

快速小结

对于长时间运行的应用来说，应该一直使用 server 编译器，最好配合分层编译。

4.3 Java和JIT编译器版本

各种测试的编译器之间是有差别的，我们来看下如何获得合适的编译器。在你下载 Java 时，需要选择版本，而最终的选择取决于你所用的平台。Java 版本的选择也会影响 JIT 编译器。到目前为止，我们讨论了 client 和 server 编译器，实际上 JIT 编译器有 3 种版本：

- 32 位 client 编译器（`-client`）
- 32 位 server 编译器（`-server`）
- 64 位 server 编译器（`-d64`）

从某种程度上说，你选择使用的编译器取决于所给的命令行选项参数（`-server` 等）。然而事情并没有这么简单。

32 位还是 64 位？

如果是 32 位操作系统，那你必须使用 32 位的 JVM。如果是 64 位操作系统，那你可以选择 32 位或 64 位 Java。并没有规定 64 位操作系统必须使用 64 位 Java。

如果堆小于 3 GB，32 位的 Java 会更快一些，并且内存占用也更少。这是因为 JVM 内部的指针只有 32 位，操作 32 位指针的代价要少于 64 位指针的（即便你使用的是 64

位 CPU）。而且 32 位指针所占的内存也少。

第 8 章讨论了压缩的普通对象指针（ordinary object pointers，oops），这是一种在 64 位 JVM 中使用 32 位寻址的方法。不过，即便有这种优化，64 位 JVM 的内存占用仍然大于 32 位 JVM，这是因为它所用的本地代码依然是 64 位寻址。

32 位 JVM 的不足之处是进程最多只能占用 4 GB 内存（某些版本的 Windows 为 3 GB，某些老版本的 Linux 为 3.5 GB）。而且还包括了堆、永久代（permgen）和本地代码以及 JVM 所用的本地内存。还有一个非常特殊的案例，因为 32 位 JVM 无法使用 CPU 的 64 位寄存器，所以大量使用 `long` 或 `double` 变量的程序在 32 位 JVM 上就会比较慢。

在 32 位 JVM 上运行的程序，只要与 32 位寻址空间吻合，无论机器是 32 位还是 64 位，都要比在类似配置的 64 位 JVM 上运行时快 5% 到 20%。比如，本章前面讨论的股票批处理程序，在我台式机的 32 位 JVM 上运行时，就要快 20%。

在下载特定操作系统的 Java 时，只有两个选项：32 位或 64 位 JVM。而准确地说，32 位 JVM（最多）有两种编译器，而 64 位只有一种编译器。（实际上 64 位 JVM 也有两种编译器，因为分层编译需要有 client 编译器的支持。但在只有 client 编译器时，64 位 JVM 无法运行。）

不过一旦安装之后，事情就变得有些复杂了。在大多数平台上，32 位和 64 位是分开安装的。在你的电脑里可以有两种 JVM，但必须通过不同路径来区分引用。因此，在我测试 Linux 的机器上，我安装了 /export/VMs/jdk1.7.0-32bit 和 /export/VMs/jdk1.7.0-64bit，通过设置相应的 `PATH` 来选择。

在 Solaris 上则有所不同：64 位的安装会覆盖 32 位。因此所有的 3 种编译器都在相同的路径上。对于最终用户来说更容易。尤其是，Java 安装在系统路径 /usr/bin 上时，用户就总能通过命令行从 3 种编译器中指定他所想要的。不过这种安装方式也有问题，由于 HotSpot 的开发人员通常使用 Solaris 作为主要的开发系统，所以事情就变得更复杂了，所用的安装方式会使讨论（有时是文档）变得令人迷惑。

关于编译器，最后要说的一点是：考虑到兼容性，指定使用哪种编译器的规则并没有被严格遵守。如果 64 位 JVM 上设定 `-client`，那无论怎样应用都会使用 64 位 server 编译器。如果 32 位 JVM 上设定 `-d64`，就会抛出所给实例不支持 64 位 JVM 的错误。

总结一下：编译器的选择取决于安装的 JVM 是 32 位还是 64 位，以及传递给 JVM 的编译器参数。表 4-4 显示了在给定 JVM 安装版本、给定参数时的编译器。

表4-4：不同JVM安装版本和不同参数下的编译器

安装的JVM版本	`-client`	`-server`	`-d64`
Linux 32 位	32 位 client 编译器	32 位 server 编译器	出错
Linux 64 位	64 位 server 编译器	64 位 server 编译器	64 位 server 编译器
Mac OS X	64 位 server 编译器	64 位 server 编译器	64 位 server 编译器
Solaris 32 位	32 位 client 编译器	32 位 server 编译器	出错
Solaris 64 位	32 位 client 编译器	32 位 server 编译器	64 位 server 编译器

（续）

安装的JVM版本	-client	-server	-d64
Windows 32 位	32 位 client 编译器	32 位 server 编译器	出错
Windows 64 位	64 位 server 编译器	64 位 server 编译器	64 位 server 编译器

在 Java 8 中，上述所有情况的默认值是 server 编译器，同时也默认开启分层编译。

不设定编译器参数会发生什么？在代码运行时，JVM 选择默认的编译器：默认编译器是运行时选择。这个选择基于 JVM 认为机器是“client”机器还是“server”机器。这个决策综合考虑了操作系统以及机器上的 CPU 数目。表 4-5 列出了不同的默认值。

表4-5：不同OS和机器上的默认编译器

操作系统	默认编译器
Windows 32 位，任意数量的 CPU	-client
Windows 64 位，任意数量的 CPU	-server
MacOS，任意数量的 CPU	-server
Linux/Solaris 32 位，1 个 CPU	-client
Linux/Solaris 32 位，2 个或以上的 CPU	-server
Linux 64 位，任意数量的 CPU	-server
Solaris 32 位 /64 位 overlay，1 个 CPU	-client
Solaris 32 位 /64 位 overlay，2 个或以上的 CPU	-server（32 位模式）

确定默认的编译器

想确定所安装的 Java 的默认编译器，可以运行以下命令：

```
% java -version
java version"1.7.0"
Java(TM) SE Runtime Environment (build 1.7.0-b147)
Java HotSpot(TM) Server VM (build 21.0-b17, mixed mode)
```

上述示例来自我的 Linux 桌面系统，32 位 Java。最后一行表示所用的编译器为以下三种之一：client（32 位）、server（32 位）或 server（64 位）。如果所安装的 Java 不支持特定的编译器，最后一行将显示实际使用的编译器：

```
% java -client -version
java version"1.7.0"
Java(TM) SE Runtime Environment (build 1.7.0-b147)
Java HotSpot(TM) 64-Bit Server VM (build 21.0-b17, mixed mode)
```

这个例子用的是 64 位 Java，在 Linux 上只支持 64 位 server 编译器。

默认值是基于一个理念的，即启动时间对 32 位 Windows 机器来说是最重要的，而基于 Unix 的系统一般来说更关注长期运行的性能。一如既往，这其中也有例外：确切来说，现代 Windows 也可以运行在高性能服务器上，即便是 32 位模式，这种情况下也应该使用 server 编译器。与此类似，许多应用服务器使用简单的 Java 管理命令侦测或变更配置，即

使在 Unix 机器上，这些命令用 client 编译器也很快。

快速小结

1. 不同的 Java 支持不同的编译器。
2. 不同的操作系统和架构所支持的编译器也不相同。
3. 程序不必指定编译器，而是仰仗平台所支持的编译器。

4.4 编译器中级调优

大多数情况下，所谓编译器调优，其实就只是为目标机器上的 Java 选择正确的 JVM 和编译器开关（`-client`、`-server` 或 `-XX:+TieredCompilation`）而已。分层编译通常是长期运行应用的最佳选择，而对于运行时间短的应用来说，分层编译与 client 编译器的性能差别也只在毫厘之内。

除了选择 JVM 和编译器开关，有些场景还需要进行额外的调优工作，本节就来探讨这一点。

4.4.1 调优代码缓存

JVM 编译代码时，会在代码缓存中保留编译之后的汇编语言指令集。代码缓存的大小固定，所以一旦填满，JVM 就不能编译更多代码了。

很显然，如果代码缓存过小，就可能会有问题。一些热点被编译了，而其他则没有，最终导致应用的大部分代码都是解释运行（非常慢）。

这个问题在使用 client 编译器或进行分层编译时很常见。使用常规的 server 编译器时，因为通常只有少量类会被编译，所以能被编译的类不太可能填满代码缓存。而用 client 编译器时，可被编译的类可能会非常多（因此也适合开启分层编译）。

代码缓存填满时，JVM（通常）会发出以下警告：

```
Java HotSpot(TM) 64-Bit Server VM warning: CodeCache is full.
        Compiler has been disabled.
Java HotSpot(TM) 64-Bit Server VM warning: Try increasing the
        code cache size using -XX:ReservedCodeCacheSize=
```

有时候容易忽略这个信息，而且开启分层编译时，Java 7 某些版本所输出的信息也不正确。本节后面将讨论另一种判断编译器是否停止编译代码的方法，即追踪输出的编译日志。

表 4-6 列出了不同平台上代码缓存的默认值。

表4-6：各种平台上代码缓存的默认大小

JVM类型	代码缓存的默认大小
32 位 client，Java 8	32 MB
32 位 server，分层编译，Java 8	240 MB
64 位 server，分层编译，Java 8	240 MB
32 位 client，Java 7	32 MB
32 位 server，Java 7	32 MB
64 位 server，Java 7	48 MB
64 位 server，分层编译，Java 7	96 MB

Java 7 开启分层编译时，默认的代码缓存通常就不够用了，常常需要扩大。使用 client 编译器的大型程序也需要增加代码缓存的大小。

确实没有什么好的机制可以算出程序所需要的代码缓存。所以，如何增加代码缓存，基本上就是摸着石头过河，通常的做法是简单地增加 1 倍或 3 倍。

`-XX:ReservedCodeCacheSize=N`（对特定编译器来说，*N* 为默认的值）标志可以设置代码缓存的最大值。代码缓存的管理和大多数 JVM 内存一样，有初始值（由 `-XX:InitialCodeCacheSize=N` 指定）。代码缓存从初始大小开始分配，一旦充满就会增加（直至最大值）。代码缓存的初始大小依据芯片架构和所用的 JVM 编译器而有所不同（例如 Intel 机器的 client 编译器的初始代码缓存为 160 KB，server 编译器的初始代码缓存为 2496 KB）。缓存大小的自动调整在后台进行，不会对性能造成实际影响，所以通常只需要设定 `ReservedCodeCacheSize`（也就是设定代码缓存的最大值）。

为了永远不超出空间而将代码缓存的最大值设得很大，这有没有什么坏处？这取决于目标机器上有多少可用资源。代码缓存设为 1 GB，JVM 就会保留 1 GB 的本地内存空间。虽然这部分内存在需要时才会分配，但它仍然是被保留的，这意味着为了满足保留内存，你的机器必须有足够的虚拟内存。

> **保留内存与已分配内存**
>
> 理解 JVM 保留内存和分配内存方式之间的差别非常重要。这种差别在代码缓存、Java 堆以及其他 JVM 本地内存结构中都存在。
>
> 关于这个主题的详细情况，请参见 8.1 节的“内存占用”。

此外，如果是 32 位 JVM，则进程占用的总内存不能超过 4 GB。这包括 Java 堆、JVM 自身所有代码占用的空间（包括它的本地库和线程栈）、分配给应用的本地内存（或者 NIO 库的直接内存），当然还有代码缓存。

鉴于上述原因，代码缓存总是受限的，大型应用（甚至使用分层编译时的中型应用）有时需要就此进行调优。然而，特别是在 64 位机器上，这个值设置得太高未必有实际效果，因为应用不可能超过进程的空间内存，且一般来说，操作系统会保留更多的内存。

通过 `jconsole`Memory（内存）面板的 Memory Pool Code Cache 图表，可以监控代码缓存。

快速小结

1. 代码缓存是一种有最大值的资源，它会影响 JVM 可运行的编译代码总量。
2. 分层编译很容易达到代码缓存默认配置的上限（特别是在 Java 7 中）。使用分层编译时，应该监控代码缓存，必要时应该增加它的大小。

4.4.2　编译阈值

本章已经粗略地定义了触发代码编译的条件。其中最主要的因素是代码执行的频度。一旦执行达到一定次数，且达到了编译阈值，编译器就可以获得足够的信息编译代码了。

本节将讨论影响这些阈值的调优标志。不过，本节实际上是为了让你对编译器如何工作有个更深入的了解（并引入一些术语）。实际上只有一种情况需要调优编译阈值，将在本节最后讨论。

编译是基于两种 JVM 计数器的：方法调用计数器和方法中的循环回边计数器。回边实际上可看作是循环完成执行的次数，所谓循环完成执行，包括达到循环自身的末尾，也包括执行了像 `continue` 这样的分支语句。

JVM 执行某个 Java 方法时，会检查该方法的两种计数器总数，然后判定该方法是否适合编译。如果适合，该方法就进入编译队列（队列的详细信息请参见 4.5.1 节的“编译线程”）。这种编译没有正式的名称，通常叫标准编译。

但是，如果循环真的很长——或因包含所有程序逻辑而永远不退出，又该如何？在这种情况下，JVM 不等方法被调用就会编译循环。所以循环每完成一轮，回边计数器就会增加并被检测。如果循环的回边计数器超过阈值，那这个循环（不是整个方法）就可以被编译。

这种编译称为栈上替换（On-Stack Replacement，OSR）。由于仅仅编译循环还不够，JVM 必须在循环进行的时候还能编译循环。在循环代码编译结束后，JVM 就会替换还在栈上的代码，循环的下一次迭代就会执行快得多的编译代码。

标准编译由 `-XX:CompileThreshold=N` 标志触发。使用 client 编译器时，*N* 的默认值是 1500，使用 server 编译器时为 10 000。更改 `CompileThreshold` 标志的值，将使编译器提早（或延后）编译。然而请注意，尽管有一个标志，但这个标志的阈值等于回边计数器加上方法调用计数器的总和。

更改 OSR 编译

更改 OSR 编译阈值的情况非常罕见。事实上，虽然 OSR 编译在基准测试（特别是微基准测试）中经常发生，但在实际运行时并不经常出现。

具体来说，OSR 编译由 3 个标志触发：

```
OSR trigger = (CompileThreshold *
        ((OnStackReplacePercentage - InterpreterProfilePercentage)/100))
```

所有编译器中的 `-XX:InterpreterProfilePercentage=N` 标志的默认值为 33。client 编译器 `-XX:OnStackReplacePercentage=N` 的默认值为 933，所以在它开始 OSR 编译前，回边计数器需要达到 13 500。在 server 编译器中，由于 `OnStackReplacePercentage` 默认值为 140，所以当回边计数器达到 10 700 时才开始 OSR 编译。注意，对于分层编译来说，虽然概念相同，但上述默认值完全取决于不同的标志。详情请看 4.5 节“高级编译器调优”

有一段时期，性能调优中常常会建议更改 `CompileThreshold` 标志。事实上，Java 的基准测试经常使用该标志（例如，常常在 server 编译器执行 8000 次迭代之后使用它）。

我们已经了解，client 编译器和 server 编译器的基本性能有很大差异，这些差异很大程度上取决于编译方法时编译器所能获得的信息。降低编译阈值，特别是对于 server 编译器来

说，可能会减少编译代码的优化——不过，应用测试表明，事实上几乎没有什么差别，比如 8000 次调用和 10 000 次调用差别甚微。

你可以认为，JVM 供应商提交的基准测试已经验证过上述调优，不同设置的基准测试间并没有什么性能差异。他们使用较低的设置主要基于以下两个原因：

- 节约一点应用热身的时间；
- 使得某些原本可能不会被 server 编译器编译的方法得以编译。

第一点应该很好理解，但第二点，为什么有些重要方法永远都不会被编译呢？并不是还没达到编译阈值，而是永远都达不到。这是因为虽然计数器随着方法和循环的执行而增加，但它们也会随时间而减少。

每种计数器的值都会周期性减少（特别是当 JVM 达到安全点时）。实际上，计数器只是方法或循环*最新*热度的度量。由此带来的一个副作用是，执行不太频繁的代码可能永远不会编译，即便是永远运行的程序（相对于热来说，有时称这些方法为温热 [lukewarm]）。这就是通过减少编译阈值来进行优化的一种情况，它也是分层编译通常比单独的 server 编译器要快的原因之一。下节将展示对于特定方法如何判定是否需要编译。如果应用分析信息显示关键路径上的方法没有编译，那有时就可以通过降低编译器阈值来触发这些方法的编译。

快速小结

1. 当方法和循环执行次数达到某个阈值的时候，就会发生编译。
2. 改变阈值会导致代码提早或推后编译。
3. 由于计数器会随着时间而减少，以至于“温热”的方法可能永远都达不到编译的阈值（特别是对 server 编译器来说）。

4.4.3 检测编译过程

关于中级优化，最后要讨论的内容并不是优化本身，这些调优措施并不会改善应用性能。更准确地说，它们就是可以让人看到编译器是如何工作的 JVM 标志（和其他工具）。其中最重要的是 `-XX:+PrintCompilation`（默认为 `false`）。

如果开启 `PrintCompilation`，每次编译一个方法（或循环）时，JVM 就会打印一行被编译的内容信息。输出的信息在不同的 Java 发布版之间会有所不同，这里的输出是 Java 7 中已经标准化的信息。

绝大多数编译日志的行具有以下格式：

```
timestamp compilation_id attributes (tiered_level) method_name size deopt
```

此处的时间戳 `timestamp` 是编译完成的时间（相对于 JVM 开始的时间 0）。

`compilation_id` 是内部的任务 ID。通常这个数字只是简单地单调增长，不过在使用 server 编译器时（或者某个时刻编译器的线程数增加时），你有时会发现乱序的 `compilation_id`。这表明编译线程相对于其他线程快或者慢了，但不能就以此下结论，某个特定的编译任务因为某种原因变得特别慢了，因为这通常只是线程调度的缘故（尽管 OSR 编译比较慢，经常出现乱序）。

attributes 是一组 5 个字符长的串，表示代码编译的状态。如果给定的编译被赋予了特定属性，就会打印下面列表中所显示的字符，否则该属性就打印一个空格。因此，5 字符属性串可以同时出现 2 个或多个字符。不同的属性如下所列。

- %：编译为 OSR。
- s：方法是同步的。
- !：方法有异常处理器。
- b：阻塞模式时发生的编译。
- n：为封装本地方法所发生的编译。

其中前 3 个可以自解释。阻塞标志在当前版本的 Java 中默认永远都不会打印，表明编译不会发生在后台（详情请参见 4.5.1 节“编译线程”）。最后，n 属性表明 JVM 生成了一些编译代码以便于调用本地方法。

如果程序没有使用分层编译的方式运行，下一个字段 tiered_level 就是空的。否则就会是数字，以表明所完成编译的级别（参见 4.7 节“分层编译级别”）。

下面一个是被编译方法（或者是被 OSR 编译的包含循环的方法）的名字，打印格式为 ClassName::method。

接下来是编译后代码的大小（单位是字节）。这是 Java 字节码的大小，不是被编译代码的大小（所以很不幸，不能用来预估代码缓存的大小）。

最后，在某些情况下，编译日志行的结尾会有一条信息，表明发生了某种逆优化，通常是“made not entrant”或“made zombie”。详情参见 4.6 节“逆优化”。

用 jstat 检测编译

编译日志需要在程序启动时开启 -XX:+PrintCompilation。如果程序启动时没有开启这个标志，你可以用 jstat 了解编译器内部的部分工作情况。

jstat 有两个有关编译器信息的标志。-compiler 标志提供了关于多少方法被编译的概要信息（此处 5003 是被检测进程的 ID）：

```
% jstat -compiler 5003
Compiled Failed Invalid   Time    FailedType FailedMethod
     206      0       0     1.97           0
```

请注意，这里也列出了编译失败的方法个数和最近编译失败的方法名。如果你通过性能分析或其他信息推测某个方法比较慢是因为没有编译，那这行命令就是一个简单验证该假设的方法。

此外，你可以用 -printcompilation 标志获取最近被编译的方法。jstat 借助一个可选参数反复执行操作，你可以看到随时间变化有哪些方法被编译了。在本例中，jstat 每 1 秒（1000 毫秒）输出一次进程 ID 为 5003 的信息：

```
% jstat -printcompilation 5003 1000
Compiled  Size  Type Method
     207     64    1 java/lang/CharacterDataLatin1 toUpperCase
     208      5    1 java/math/BigDecimal$StringBuilderHelper getCharArray
```

编译日志还会包括类似下面这行信息：

```
timestamp compile_id COMPILE SKIPPED: reason
```

这行信息（包括文本文字 COMPILE SKIPPED）表示编译给定的方法有错误。出现这个错可能有以下两种原因。

代码缓存满了

需用 ReservedCodeCache 标志增大代码缓存的大小。

编译的同时加载类

编译类的时候会发生修改。JVM 之后会再次编译，你可以在之后的日志中看到方法被再次编译。

在所有这些情况（除了代码缓存被填满）中，编译都可以再次尝试。如果不能，说明代码编译出了错。虽然通常是编译器的缺陷，但常用的解决方法是将代码重构得更简单，以使编译器能够处理。

以下是股票的 servlet Web 应用开启 PrintCompilation 时的几行输出：

```
 28015  850           net.sdo.StockPrice::getClosingPrice (5 bytes)
 28179  905  s        net.sdo.StockPriceHistoryImpl::process (248 bytes)
 28226   25 %         net.sdo.StockPriceHistoryImpl::<init> @ 48 (156 bytes)
 28244  935           net.sdo.MockStockPriceEntityManagerFactory$\
                          MockStockPriceEntityManager::find (507 bytes)
 29929  939           net.sdo.StockPriceHistoryImpl::<init> (156 bytes)
106805 1568   !       net.sdo.StockServlet::processRequest (197 bytes)
```

输出包括仅有的一些与股票相关的已编译方法。几个有趣的地方值得注意：首个与股票相关的方法在应用服务器启动 28 秒之后才被编译，在它之前有 849 个被编译的方法。在这个例子中，这些都是应用服务器的方法（从输出日志中可以过滤出来）。应用服务器启动用了 2 秒，剩下的没开始编译之前的 26 秒基本上是空闲，因为应用服务器在等待请求。

输出中剩余的几行显示了一些重要特性。process() 是同步方法，这和你在代码列表中所见到的一样。内部类的编译和其他类一样，在日志中遵循 Java 的命名规则：outer-classname$inner-classname。不出所料，processRequest() 有异常处理器。

最后，回顾一下 StockPriceHistoryImpl 的构造函数，它包含了一个大循环：

```
public StockPriceHistoryImpl(String s, Date startDate, Date endDate) {
    EntityManager em = emf.createEntityManager();
    Date curDate = new Date(startDate.getTime());
    symbol = s;
    while (!curDate.after(endDate)) {
         StockPrice sp = em.find(StockPrice.class, new StockPricePK(s, curDate));
         if (sp != null) {
            if (firstDate == null) {
                firstDate = (Date) curDate.clone();
            }
            prices.put((Date) curDate.clone(), sp);
            lastDate = (Date) curDate.clone();
         }
```

```
            curDate.setTime(curDate.getTime() + msPerDay);
        }
    }
```

这个循环的执行次数比构造函数本身多，所以是 OSR 编译的目标。请注意，编译构造函数花费了一点时间，开始时它的编译 ID 为 25，但直到编译其他方法又过了 900 多个编译 ID 之后才再次出现该方法。（这个例子中的 OSR 行信息容易被误读成 25%，你会好奇其他 75% 是什么。请注意，这里的数字只是编译 ID，而 % 只表示 OSR 编译。）这是典型的 OSR 编译，栈上替换比较困难，期间还会持续进行其他编译。

快速小结

1. 观察代码如何被编译的最好方法是开启 `PrintCompilation`。
2. `PrintCompilation` 开启后所输出的信息可用来确认编译是否和预期一样。

4.5 高级编译器调优

本节将补充一些编译如何工作的细节，在此过程中探索一些可以影响编译的调优方法。不过，虽然可以更改这些值，但真的不建议这么做，因为这些调优标志很大程度上是为了帮助 JVM 工程诊断 JVM 行为的。如果你对编译器的工作原理非常好奇，那你会对本节感兴趣，如果不是，可直接跳过本节内容。

4.5.1 编译线程

4.4.2 节“编译阈值”曾提到，当方法（或循环）适合编译时，就会进入到编译队列。队列则由一个或多个后台线程处理。这是件好事，意味着编译过程是异步的；这使得即便是代码正在编译的时候，程序也能持续执行。如果是用标准编译所编译的方法，那下次调用该方法时就会执行编译后的方法；如果是用 OSR 编译的循环，那下次循环迭代时就会执行编译后的代码。

编译队列并不严格遵守先进先出的原则：调用计数次数多的方法有更高的优先级。所以，即便在程序开始执行并有大量代码需要编译时，这样的优先顺序仍然有助于确保最重要的代码优先编译。（这是为何 `PrintCompilation` 输出中的 `ID` 为乱序的另一个原因。）

当使用 client 编译器时，JVM 会开启一个编译线程；使用 server 编译器时，则会开启两个这样的线程。当启用分层编译时，JVM 默认开启多个 client 和 server 线程，线程数依据一个略复杂的等式而定，包括目标平台 CPU 数取双对数之后的数值。表 4-7 中显示的值即为计算出的数值。

表4-7：分层编译中编译器C1和C2的默认线程数

CPU数量	C1的线程数	C2的线程数
1	1	1
2	1	1
4	1	2
8	1	2

（续）

CPU数量	C1的线程数	C2的线程数
16	2	6
32	3	7
64	4	8
128	4	10

编译器的线程数（3 种编译器都是如此）可通过 `-XX:CICompilerCount=N` 标志来设置（默认值参见前表）。这是 JVM 处理队列的线程总数；对分层编译来说，其中三分之一（至少一个）将用来处理 client 编译器队列，其余的线程（至少一个）用来处理 server 编译器队列。

你何时需要考虑调整该参数值？如果程序运行在单 CPU 系统上，那么只有设置成单个编译器线程才可以得到些好处：对可用 CPU 受限的系统来说，在许多情况下只有减少争抢资源的线程数才有利于性能提升。但是，这种好处也仅限于初始的热身阶段；在此之后，已编译过的方法将不会再引起 CPU 竞争。当股票配比处理应用运行在单 CPU 机器上，并且编译器线程数限制为一时，初始计算会快大约 10%（因为不用经常争抢 CPU）。运行的轮次越多，初始时的整体收益就越小，直到所有热点方法都被编译之后，这种收益就消失了。

使用分层编译时，线程数很容易超过系统限制，特别是有多个 JVM 同时运行的时候（每个都开启很多编译线程）。在这种情况下，减少线程数有助于提升整体的吞吐量（尽管代价可能是热身期会持续得更长）。

与此类似，如果有额外可用的 CPU 周期，理论上程序将会受益——至少在热身期间——此时编译器线程数会增加。在实际工作中，这样的好处很难获得。进一步来说，如果有很多可用的 CPU，那么在应用的整个执行过程中，你都可以去尝试那些能充分发挥可用 CPU 周期的方法（而不仅仅在开始时加快编译），这样会好得多。

另外一个编译线程的设定参数是 `-XX:+BackgroundCompilation` 标志，默认值为 `true`。这意味着，和参数所描述的一样，编译队列的处理是异步执行的。但这个参数也可以设置为 `false`，在这种情况下，当一个方法适合编译，执行该方法的代码将一直等到它确实被编译之后才执行（而不是继续在解释器中执行）。用 `-Xbatch` 可以禁止后台编译。

快速小结

1. 放置在编译队列中的方法的编译会被异步执行。
2. 队列并不是严格按照先后顺序的；队列中的热点方法会在其他方法之前编译。这是编译输出日志中的 ID 为乱序的另一个原因。

4.5.2　内联

编译器所做的最重要的优化是方法内联。遵循面向对象设计的良好代码通常都会包括一些需要通过 getter（也可能包含 setter）访问的属性：

```
public class Point {
    private int x, y;
```

```
        public void getX() { return x; }
        public void setX(int i)  { x = i; }
    }
```

此类方法调用的开销很大，特别是相对于方法的代码量而言。事实上，在早期的 Java 中，考虑到所有此类调用对性能的影响，性能调优小贴士常常会信誓旦旦地反对此类封装。幸运的是，现在的 JVM 通常都会用内联代码的方式执行这些方法。因此，你可以这样写代码：

```
Point p = getPoint();
p.setX(p.getX() * 2);
```

而编译后的代码本质上执行的是：

```
Point p = getPoint();
p.x = p.x * 2;
```

内联默认是开启的。可通过 `-XX:-Inline` 关闭，然而由于它对性能的影响巨大，事实上你永远不会这么做（例如，关闭内联的话，股票配比测试的性能会减少 50%）。由于内联非常重要，并且还因为有许多其他控制标志，所以通常都会建议对 JVM 内联进行调优。

不幸的是，基本上没法看到 JVM 是如何内联代码的。（如果你从源代码编译 JVM，那可以用 `-XX:+PrintInlining` 生成带调试信息的版本。这个参数会提供所有关于编译器如何进行内联决策的信息。）最好的方法是查看代码的分析信息，如果在分析信息的顶部附近有简单的方法，并且看起来这些方法应该内联，可用内联标志做些试验。

方法是否内联取决于它有多热以及它的大小。JVM 依据内部计算来判定方法是否是热点（譬如，调用很频繁）；是否是热点并不直接与任何调优参数相关。如果方法因调用频繁而可以内联，那只有在它的字节码小于 325 字节时（或 `-XX:MaxFreqInlineSize=N` 所设定的任意值）才会内联。否则，只有方法很小时，即小于 35 字节（或 `-XX:MaxInlineSize=N` 所设定的任意值）时才会内联。

有时你会看到增加 `MaxInlineSize` 的值以便内联更多方法的建议。两者之间常被忽略的是，`MaxInlineSize` 超过 35 意味着第一次调用方法时就会被内联。然而，方法只有经常被调用时——在这种情况下它的性能会受更大影响——最终才值得内联（假定它的大小小于 325 字节）。否则，`MaxInlineSize` 调优的最终结果就是减少了热身测试所需的时间，但不太可能对长期运行的程序产生重大影响。

快速小结

1. 内联是编译器所能做的最有利的优化，特别是对属性封装良好的面向对象的代码来说。
2. 几乎用不着调节内联参数，且提倡这样做的建议往往忽略了常规内联和频繁调用内联之间的关系。当考察内联效应时，确保考虑这两种情况。

4.5.3 逃逸分析

如果开启逃逸分析（`-XX:+DoEscapeAnalysis`，默认为 `true`），server 编译器将会执行一些非

常激进的优化措施。比如，考虑以下计算阶乘的类：

```
public class Factorial {
    private BigInteger factorial;
    private int n;
    public Factorial(int n) {
        this.n = n;
    }
    public synchronized BigInteger getFactorial() {
        if (factorial == null)
            factorial = ...;
        return factorial;
    }
}
```

若想在数组中保存前 100 个阶乘值，使用以下代码：

```
ArrayList<BigInteger> list = new ArrayList<BigInteger>();
for (int i = 0; i < 100; i++) {
    Factorial factorial = new Factorial(i);
    list.add(factorial.getFactorial());
}
```

`factorial` 对象只在循环中引用；没有任何其他代码可以访问该对象。因此，JVM 会毫不犹豫地对这个对象进行一系列优化。

- 当调用 `getFactorial()` 时，没必要获得同步锁。
- 没必要在内存中保存 `n`；可以在寄存器中保存该值。同样，`factorial` 也可以保存在寄存器中。
- 事实上，根本就不需要分配实际的 `factorial` 对象；可以只追踪这个对象的个别字段。

此类优化非常复杂：虽然这个例子非常简单，但此类优化可能会伴随更复杂的代码。由于所用的代码不同，并不是所有的优化都有必要使用。但逃逸分析可以决定哪些优化是可能的，并决定编译后的代码中哪些是必要的改变。

逃逸分析默认开启。极少数情况下，它会出错，在此类情况下关闭它会变得更快或更稳定。如果你发现了这种情况，最好的应对行为就是简化相关代码：代码越简单越好。（不过如果是 bug，则应该发送报告。）

快速小结

1. 逃逸分析是编译器能做得最复杂的优化。此类优化常常会导致微基准测试失败。
2. 逃逸分析常常会给不正确的同步代码引入“bug”。

4.6　逆优化

`PrintCompilation` 标志输出的讨论中曾提到两种代码逆优化的情况。逆优化意味着编译器不得不“撤销”之前的某些编译；结果是应用的性能降低——至少是直到编译器重新编译相应代码为止。

有两种逆优化的情形：代码状态分别为“made not entrant”（代码被丢弃）和“made zombie”（产生僵尸代码）时。

4.6.1 代码被丢弃

有两种原因导致代码被丢弃。可能是和类与接口的工作方式有关，也可能与分层编译的实现细节有关。

先考虑第一种情况。回想一下 stock 应用的接口 `StockPriceHistory`。示例代码中有两种接口实现：基本实现（`StockPriceHistoryImpl`）和每个操作带有日志的实现（`StockPriceHistoryLogger`）。servlet 代码依据请求 URL 上的 `log` 参数来选择实现：

```
StockPriceHistory sph;
String log = request.getParameter("log");
if (log != null && log.equals("true")) {
    sph = new StockPriceHistoryLogger(...);
}
else {
    sph = new StockPriceHistoryImpl(...);
}
// 然后JSP调用：
sph.getHighPrice();
sph.getStdDev();
// 等等
```

如果向 http://localhost:8080/StockServlet 发起一组请求调用（没有 `log` 参数），那么编译器会看到 `sph` 的实际类型为 `StockPriceHistoryImpl`。然后它将内联代码，并据此执行其他优化。

之后再向 http://localhost:8080/StockServlet?log=true 发起一次调用。现在编译器依据 `sph` 类型所做的假定就不成立了，之前的优化也失效了。这就产生了逆优化陷阱（deoptimization trap），之前的优化也被废弃了。如果有更多这样带有 `log=true` 的请求调用，JVM 就会很快终止这部分代码编译，而开始新的编译。

该场景的编译日志包括了以下信息：

```
 841113   25 %          net.sdo.StockPriceHistoryImpl::<init> @ -2 (156 bytes)
                                made not entrant
 841113  937  s         net.sdo.StockPriceHistoryImpl::process (248 bytes)
                                made not entrant
1322722   25 %          net.sdo.StockPriceHistoryImpl::<init> @ -2 (156 bytes)
                                made zombie
1322722  937  s         net.sdo.StockPriceHistoryImpl::process (248 bytes)
                                made zombie
```

请注意，OSR 编译过的构造函数和标准编译过的方法都被标记成 `made not entrant`，过了一会，它们又被标记成 `made zombie`。

逆优化听起来不太好，至少作为性能优化的术语是这样，但并非总是如此。本章的第一个例子中用了股票的 servlet 应用，只测了触发 `StockPriceHistoryImpl` 的 URL。测试热身时间 300 秒，使用分层编译，达到 24.4 OPS。

假定上述测试之后立刻用 `StockPriceHistoryLogger` 测试——也就是我刚刚列举的那个逆

优化例子。PrintCompilation 的完整输出显示，请求带日志的 StockPriceHistory 实现时，StockPriceHistoryImpl 类的所有方法都被逆优化了。不过，逆优化之后，如果再次请求 StockPriceHistoryImpl，这些代码又会重新编译（原先的假设会发生少许差异），最终我们仍将看到大约 24.4 OPS（在新一轮热身之后）。

当然这是最好的情况。如果混合这些调用，使得编译器无法假定采用哪种代码路径，会发生什么？因为有额外的日志，所以通过 servlet 访问带日志的路径大约需要 24.1 OPS。如果混合操作，大约为 24.3 OPS，与期望的平均值相近。在批处理程序中也能观察到类似的结果。所以，除了进入陷阱的短暂时间，逆优化对其他方面没有产生什么重大影响。

第二种导致代码被丢弃的原因是分层编译。在分层编译中，代码先由 client 编译器编译，然后由 server 编译器编译（实际上要比这复杂一些，下一节会进一步讨论）。当 server 编译器编译好代码之后，JVM 必须替换 client 编译器所编译的代码。它会将老代码标记为废弃，也用同样的办法替换新编译（和更有效）的代码。因此，当程序使用分层编译时，编译日志就会显示许多被丢弃的方法。不要慌张，这种“逆优化”事实上使代码变得更快了。

可以通过观察编译日志中的层次级别信息来检测逆优化：

```
40915   84 %     3       net.sdo.StockPriceHistoryImpl::<init> @ 48 (156 bytes)
40923 3697       3       net.sdo.StockPriceHistoryImpl::<init> (156 bytes)
41418   87 %     4       net.sdo.StockPriceHistoryImpl::<init> @ 48 (156 bytes)
41434   84 %     3       net.sdo.StockPriceHistoryImpl::<init> @ -2 (156 bytes)
                                     made not entrant
41458 3749       4       net.sdo.StockPriceHistoryImpl::<init> (156 bytes)
41469 3697       3       net.sdo.StockPriceHistoryImpl::<init> (156 bytes)
                                     made not entrant
42772 3697       3       net.sdo.StockPriceHistoryImpl::<init> (156 bytes)
                                     made zombie
42861   84 %     3       net.sdo.StockPriceHistoryImpl::<init> @ -2 (156 bytes)
                                     made zombie
```

这里，构造方法首先是级别 3 的 OSR 编译，然后在级别 3 得到完整编译。很快，OSR 代码变得适合级别 4 的编译，所以 JVM 在级别 4 上编译代码，而原先级别 3 的代码被丢弃。相同的过程在标准编译中也会发生，而级别 3 的编译代码最后会变成僵尸代码。

4.6.2　逆优化僵尸代码

编译日志显示产生了僵尸代码，意思是说 JVM 已经回收了之前被丢弃的代码。在上面的例子中，测试过 StockPriceHistoryLogger 之后，StockPriceHistoryImpl 类的编译代码就被丢弃了。但是 StockPriceHistoryImpl 类的对象仍然存在。最终，所有这些对象都会被 GC 回收。全部回收之后，编译器就会注意到，这个类现在适合标记为僵尸代码了。

从性能角度来看，这是好事。回想一下，编译代码保存在有固定大小的代码缓存中。如果发现僵尸方法，这意味着这些有问题的代码可以从代码缓存中移除，腾出空间给其他将被编译的代码（或者限制 JVM 之后需要分配的内存量）。

可能产生的不足是，如果代码被僵尸化以后被再次加载并且频繁使用，JVM 就需要重新编译和重新优化代码。而且情形就像上面所描述的那样，测试一会没日志，一会有日志，然

后又没有日志。这种情况下，性能并没有受到明显的影响。一般来说，像僵尸代码重编译这样小的操作对大多数应用都不会有显著的影响。

快速小结

1. 逆优化使得编译器可以回到之前版本的编译代码。
2. 先前的优化不再有效时（例如，所涉及的对象类型发生了更改），才会发生代码逆优化。
3. 代码逆优化时，会对性能产生一些小而短暂的影响，不过新编译的代码会尽快地再次热身。
4. 分层编译时，如果代码之前由 client 编译器编译而现在由 server 编译器优化，就会发生逆优化。

4.7 分层编译级别

程序使用分层编译时，编译日志中会输出代码所编译的分层级别。上一节的示例中，代码最多编译到级别 4，甚至为了简化讨论，到目前为止，我已经说过只有两种编译器（加上解释器）。

因为 client 编译器有 3 种级别，所以总共有 5 种执行级别。因此，编译级别有：

- 0：解释代码
- 1：简单 C1 编译代码
- 2：受限的 C1 编译代码
- 3：完全 C1 编译代码
- 4：C2 编译代码

典型的编译日志可以显示，多数方法第一次编译的级别是 3，即完全 C1 编译。（当然，所有方法都从级别 0 开始。）如果方法运行得足够频繁，它就会编译成级别 4（级别 3 的代码就会被丢弃）。最常见的情况是：client 编译器从获取了代码如何使用的信息进行优化时才开始编译。

如果 server 编译器队列满了，就会从 server 队列中取出方法，以级别 2 进行编译，在这个级别上，C1 编译器使用方法调用计数器和回边计数器（但不需要性能分析的反馈信息）。这使得方法编译得更快，而方法也将在 C1 编译器收集分析信息之后被编译为级别 3，最终当 server 编译器队列不太忙的时候被编译为级别 4。

另一方面，如果 client 编译器全忙，原本排程在级别 3 编译的方法就既可以等待级别 3 编译，也适合进行级别 4 的编译。在这种情况下，方法编译会很快转到级别 2，然后由级别 2 转到级别 4。

那些不太重要的方法可以从级别 2 或级别 3 开始编译，但随后会因为它们的重要性没那么高而转为级别 1。另外，如果 server 编译器出于某些原因无法编译代码，也会转为级别 1。

当然，代码在逆编译时会转为级别 0。

有些标志可以控制某些级别转换行为，但调优能够得到很乐观的结果。当方法按期望的顺序，即级别 0 → 级别 3 → 级别 4 编译时，性能可以达到最优。如果方法经常被编译为级别 2，并且还额外有可用的 CPU 周期，那就可以考虑增加编译器的线程数，从而减少 server 编译器队列的长度。如果没有额外可用的 CPU 周期，那你唯一能做的就是尽力减小应用的大小。

快速小结

1. 分层编译可以在两种编译器和 5 种级别之间进行。
2. 不建议人为更改级别，本节仅仅是辅助解释编译日志的输出。

4.8 小结

本节提供了大量关于即时编译如何工作的细节。从调优角度看，简单的选择就是对所有应用都使用 server 编译器和分层编译，这将解决 90% 的与编译器相关的性能问题。只要确保代码缓存足够大，编译器就能提供尽善尽美的性能。

关于编译器调优的最后一句话

如果你有 Java 性能调优方面的经验，你或许会觉得惊奇，因为整章关于编译的讨论中都没有提及 `final` 关键字。`final` 总被认为是影响性能的重要因素，因为大家相信在进行内联和其他优化时，`final` 可以使 JIT 编译器作出更好的选择。

这种想法在落伍的过去或许有一些价值，但已经很多年很多年不是这样了（即便曾经是）。而其实它是流传甚广的谣言。准确地说，只要有必要时，你就应该使用 `final`：比如你不打算改变的不可变对象或原生值，内部类引用的外部参数等等。但无论有没有 `final` 关键字，都不会影响应用的性能。

本章也包括了大量关于编译器如何运作的背景知识。这么做的原因是，你可以理解第 1 章关于小方法和小代码的一般性建议，以及第 2 章所描述的微基准测试对编译器的影响。特别是，还涵盖了以下知识。

(1) 不用担心小方法——特别是 getter 和 setter，因为它们很容易内联。如果你觉得某个方法的负载过大，那你可能只是在理论上是正确的（通过移除内联给性能造成的巨大影响可以展示这点）。而实际上并不是这样，因为编译器会修复这些问题。

(2) 需要编译的代码在编译队列中。队列中代码越多，程序达到最佳性能的时间越久。

(3) 虽然代码缓存的大小可以（也应该）调整，但它仍然是有限的资源。

(4) 代码越简单，优化越多。分析反馈和逃逸分析可以使代码更快，但复杂的循环结构和大方法限制了它的有效性。

最后，如果你在分析代码时，很惊奇地发现一些方法出现在了分析信息的顶部——你原本期望不应该出现在那里——你可以使用本章的内容审视编译器正在做什么，以确保它正在以你所写的代码逻辑处理。

第 5 章

垃圾收集入门

这一章我们会一起探究 JVM 垃圾收集的基础知识。很多时候我们没有机会重写代码，又面临需要提高 Java 应用性能的压力，这种情况下对垃圾收集器的调优就变得至关重要。

现代 JVM 的类型繁多，最主流的四个垃圾收集器分别是：Serial 收集器（常用于单 CPU 环境）、Throughput（或者 Parallel）收集器、Concurrent 收集器（CMS）和 G1 收集器。它们的性能特征迥异，下一章将围绕每种垃圾收集器的特质进行深入讨论。虽然存在差异，不过它们也有不少共性，本章会针对这些共性概述垃圾收集器是如何工作的。

5.1 垃圾收集概述

对程序员而言，Java 最诱人的特性之一是不需要显式地管理对象的生命周期：我们可以在需要时创建对象，对象不再被使用时，会由 JVM 在后台自动进行回收。但是，如果你像我一样，花费大量的时间来优化 Java 程序的内存使用，这套机制看起来可能更像个缺点，而非其优点（我在垃圾收集调优上耗费的时间似乎也证明了这一点）。当然，这其实是一把双刃剑。不过，在过去 15 年里，我在 Java 内存问题处理上花费的时间，远少于过去 10 年里（使用其他语言时）查找并修复因指针悬挂和空指针引起的问题所消耗的时间。

简单来说，垃圾收集由两步构成：查找不再使用的对象，以及释放这些对象所管理的内存。JVM 从查找不再使用的对象（垃圾对象）入手。有时，这也被称为查找不再有任何对象引用的对象（暗指采用“引用计数”的方式统计对象引用）。然而，这种靠引用计数的方式不太靠谱：假设有一个对象链接列表，列表中的每一个对象（除了头节点）都指向列表中的另一个对象，但是，如果没有任何一个引用指向列表头，这个列表就没人使用，可以被垃圾回收器回收。如果这是一个循环列表（即列表的尾元素反过来又指向了头元素），那么列表中的每一个元素都包含一个引用，即使这个列表内没有任何一个对象实际被使用，因为没有任何一个对象指向这个列表。

所以引用是无法通过计数的方式动态跟踪的；JVM 必须定期地扫描堆来查找不再使用的对象。一旦发现垃圾对象，JVM 会回收这些对象所持有的内存，把它们分配给需要内存的其他对象。然而，简单地记录空闲内存也无法保证将来有足够的可用内存，有些时候，我们还必须进行内存整理来防止内存碎片。

假设以下场景，一个程序需要分配大小为 1000 字节的数组，紧接着又分配一个大小为 24 字节的数组，并在一个循环中持续进行这样的分配。最终程序会耗尽整个堆，结果如图 5-1 中的第一行所示：堆空间被占满，分配的数组间隔地分布于整个堆内。

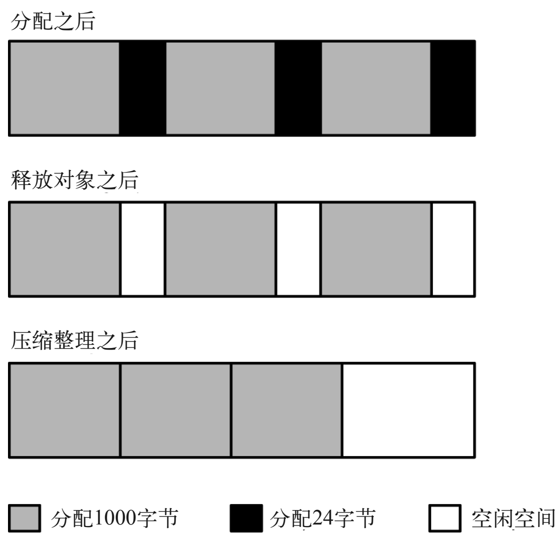


图 5-1：垃圾收集过程中理想化的 GC 堆

堆内存用尽会触发 JVM 回收不再使用的数组空间。假设所有大小为 24 字节的数组都不再被使用，而大小为 1000 字节的数组还继续使用，这就形成了图 5-1 中第二行的场景。虽然堆内部有足够的空闲空间，却找不到任何一个大于 24 字节的连续空间，除非 JVM 移动所有大小为 1000 字节的数组，让它们连续存储，把空闲的空间整合成一块更大的连续空间，供其他的内存分配使用（如图 5-1 的第三行）。

深入到这些具体实现似乎太过于琐屑，但垃圾收集的性能就是由这些基本操作所决定的：找到不再使用的对象、回收它们使用的内存、对堆的内存布局进行压缩整理。完成这些操作时不同的收集器采用了不同的方法，这也是不同垃圾收集器表现出不同性能特征的原因。

如果垃圾收集进行时，没有任何应用程序线程在运行，那么完成这些操作将是一件轻松愉快的事情。然而实际情况要复杂得多，通常 Java 程序都启动了大量的线程，垃圾收集器自身往往也是多线程的。接下来的讨论中，我们从逻辑上将线程分成了两组，分别是应用程序线程和处理垃圾收集的线程。垃圾收集器回收对象，或者在内存中移动对象时，必须确保应用程序线程不再继续使用这些对象。这一点在收集器移动对象时尤其重要：在操作过程中，对象的内存地址会发生变化，因此这个过程中任何应用线程都不应再访问该对象。

所有应用线程都停止运行所产生的停顿被称为时空停顿（stop-the-world）。通常这些停顿对应用的性能影响最大，调优垃圾收集时，尽量减少这种停顿是最为关键的考量因素。

5.1.1 分代垃圾收集器

虽然实现的细节千差万别，但所有的垃圾收集器都遵循了同一个方式，即根据情况将堆划分成不同的代（Generation）。这些代被称为“老年代”（Old Generation 或 Tenured Generation）和“新生代”（Young Generation）。新生代又被进一步地划分为不同的区段，分别称为 Eden 空间和 Survivor 空间（不过 Eden 有时会被错误地用于指代整个新生代）。

采用分代机制的原因是很多对象的生存时间非常短。譬如下面这个例子，这是一个计算股价的循环，它将股价与股票均价的差值进行乘方运算，然后将结果加和（作为标准偏差计算的一部分）。

```
sum = new BigDecimal(0);
for (StockPrice sp : prices.values()) {
    BigDecimal diff = sp.getClosingPrice().subtract(averagePrice);
    diff = diff.multiply(diff);
    sum = sum.add(diff);
}
```

BigDecimal 同许多 Java 类一样是不可变对象：该对象代表的是一个不能修改的数字。运算使用这个对象时，会创建一个新的对象（通常，前一个对象及其值会被丢弃）。这个简单的循环处理一年的股票数据（大约 250 个循环）时，为了存储循环的中间值，会创建 750 个 `BigDecimal` 对象，只是在这一个循环里。这些对象在循环的下一个周期开始时会被丢弃。在 `add()` 以及其他方法内，JDK 的库方法会创建更多类似 `BigDecimal`（以及其他的类）的中间对象。最终，在一小段代码中大量的对象被快速地创建和丢弃。

Java 中，这种操作是非常普遍的，所以垃圾收集器设计时就特别考虑要处理大量（有时候是大多数）的临时对象。这也是分代设计的初衷之一。新生代是堆的一部分，对象首先在新生代中分配。新生代填满时，垃圾收集器会暂停所有的应用线程，回收新生代空间。不再使用的对象会被回收，仍然在使用的对象会被移动到其他地方。这种操作被称为 Minor GC。

采用这种设计有两个性能上的优势。其一，由于新生代仅是堆的一部分，与处理整个堆相比，处理新生代的速度更快。而这意味着应用线程停顿的时间会更短。你可能也看到了这其中的权衡，这意味着应用程序线程会更频繁地发生停顿，因为 JVM 不再等到整个堆都填满才进行垃圾收集；本章后续部分会针对其利弊进行深入的讨论。然而，就目前而言，更短的停顿显然能带来更多的优势，即使发生的频率更高。

第二个优势源于新生代中对象分配的方式。对象分配于 Eden 空间（占据了新生代空间的绝大多数）。垃圾收集时，新生代空间被清空，Eden 空间中的对象要么被移走，要么被回收；所有的存活对象要么被移动到另一个 Survivor 空间，要么被移动到老年代。由于所有的对象都被移走，相当于新生代空间在垃圾收集时自动地进行了一次压缩整理。

所有的垃圾收集算法在对新生代进行垃圾回收时都存在“时空停顿”现象。

对象不断地被移动到老年代，最终老年代也会被填满，JVM 需要找出老年代中不再使用的对象，并对它们进行回收。而这便是垃圾收集算法差异最大的地方。简单的垃圾收集算

法直接停掉所有的应用线程，找出不再使用的对象，对其进行回收，接着对堆空间进行整理。这个过程被称为 Full GC，通常导致应用程序线程长时间的停顿。

另一方面，通过更复杂的计算，我们还有可能在应用线程运行的同时找出不再使用的对象；CMS 和 G1 收集器就是通过这种方式进行垃圾收集的。由于它们不需要停止应用线程就能找出不再用的对象，CMS 和 G1 收集器被称为 Concurrent 垃圾收集器。同时，由于它们将停止应用程序的可能降到了最小，也被称为低停顿（Low-Pause）收集器（有时也称为无停顿收集器，虽然这个叫法相当不确切）。Concurrent 收集器也使用各种不同的方法对老年代空间进行压缩。

使用 CMS 或 G1 收集器时，应用程序经历的停顿会更少（也更短）。其代价是应用程序会消耗更多的 CPU。CMS 和 G1 收集也可能遭遇长时间的 Full GC 停顿（尽量避免发生那样的停顿是这些调优算法要考虑的重要方面）。

评估垃圾收集器时，想想你需要达到的整体性能目标。每一个决定都需要权衡取舍。如果应用对单个请求的响应时间有要求（譬如 Java 企业版服务器），你应该考虑下面这些因素。

- 单个请求会受停顿时间的影响——不过其受 Full GC 长时间停顿的影响更大。如果目标是要尽可能地缩短响应时间，那么选择使用 Concurrent 收集器更合适。
- 如果平均响应时间比最大响应时间更重要（譬如 90% 的响应时间），采用 Throughput 收集器通常就能满足要求。
- 使用 Concurrent 收集器来避免长的停顿时间也有其代价，这会消耗额外的 CPU。

类似地，为批量应用选择垃圾收集器可以遵循下面的原则。

- 如果 CPU 足够强劲，使用 Concurrent 收集器避免发生 Full GC 停顿可以让任务运行得更快。
- 如果 CPU 有限，那么 Concurrent 收集器额外的 CPU 消耗会让批量任务消耗更多的时间。

快速小结

1. 所有的 GC 算法都将堆划分成了老年代和新生代。
2. 所有的 GC 算法在清理新生代对象时，都使用了“时空停顿”（stop-the-world）方式的垃圾收集方法，通常这是一个能较快完成的操作。

5.1.2 GC算法

JVM 提供了以下 4 种不同的垃圾收集算法。

1. Serial垃圾收集器

Serial 垃圾收集器是四种垃圾收集器中最简单的一种。如果应用运行在 Client 型虚拟机（Windows 平台上的 32 位 JVM 或者是运行在单处理器机器上的 JVM）上，这也是默认的垃圾收集器。

Serial 收集器使用单线程清理堆的内容。使用 Serial 收集器，无论是进行 Minor GC 还是 Full GC，清理堆空间时，所有的应用线程都会被暂停。进行 Full GC 时，它还会对老年代空间的对象进行压缩整理。通过 `-XX:+UseSerialGC` 标志可以启用 Serial 收集器（大多数情

况下，如果可以使用这个标志，默认就会开启）。注意，跟大多数的 JVM 标志不同，关闭 Serial 收集器不能简单地将加号符变成减号符（譬如，使用 `-XX:-UseSerialGC`）。在 Serial 收集器作为默认收集器的系统上，如果需要关闭 Serial 收集器，可以通过指定另一种垃圾收集器来实现。

2. Throughput垃圾收集器

Throughput 收集器是 Server 级虚拟机（多 CPU 的 Unix 机器以及任何 64 位虚拟机）的默认收集器。

Throughput 收集器使用多线程回收新生代空间，Minor GC 的速度比使用 Serial 收集器快得多。处理老年代时 Throughput 收集器也能使用多线程方式。这已经是 JDK 7u4 及之后的版本的默认行为，对于之前老版本的 JDK 7 虚拟机，通过 `-XX:+UseParallelOldGC` 标志可以开启这个功能。由于 Throughput 收集器使用多线程，Throughput 收集器也常常被称为 Parallel 收集器。Throughput 收集器在 Minor GC 和 Full GC 时会暂停所有的应用线程，同时在 Full GC 过程中会对老年代空间进行压缩整理。由于在大多数适用的场景，它已经是默认的收集器，所以你基本上不需要显式地启用它。如果需要，可以使用 `-XX:+UseParallelGC`、`-XX:+UseParallelOldGC` 标志启用 Throughput 收集器。

3. CMS收集器

CMS 收集器设计的初衷是为了消除 Throughput 收集器和 Serial 收集器 Full GC 周期中的长时间停顿。CMS 收集器在 Minor GC 时会暂停所有的应用线程，并以多线程的方式进行垃圾回收。然而，这其中最显著的不同是，CMS 不再使用 Throughput 的收集算法（`-XX:+UseParallelGC`），改用新的算法来收集新生代对象（使用 `-XX:+UseParNewGC` 标志）。

CMS 收集器在 Full GC 时不再暂停应用线程，而是使用若干个后台线程定期地对老年代空间进行扫描，及时回收其中不再使用的对象。这种算法帮助 CMS 成为一个低延迟的收集器：应用线程只在 Minor GC 以及后台线程扫描老年代时发生极其短暂的停顿。应用程序线程停顿的总时长与使用 Throughput 收集器比起来短得多。

这里额外付出的代价是更高的 CPU 使用：必须有足够的 CPU 资源用于运行后台的垃圾收集线程，在应用程序线程运行的同时扫描堆的使用情况。除此之外，后台线程不再进行任何压缩整理的工作，这意味着堆会逐渐变得碎片化。如果 CMS 的后台线程无法获得完成他们任务所需的 CPU 资源，或者如果堆变得过度碎片化以至于无法找到连续空间分配对象，CMS 就蜕化到 Serial 收集器的行为：暂停所有应用线程，使用单线程回收、整理老年代空间。这之后又恢复到并发运行，再次启动后台线程（直到下一次堆变得过度碎片化）。通过 `-XX:+UseConcMarkSweepGC`、`-XX:+UseParNewGC` 标志（默认情况下，这两个标志都是禁用的）可以启用 CMS 垃圾收集器。

4. G1垃圾收集器

G1 垃圾收集器（或者垃圾优先收集器）的设计初衷是为了尽量缩短处理超大堆（大于 4 GB）时产生的停顿。G1 收集算法将堆划分为若干个区域（Region），不过它依旧属于分代收集器。这些区域中的一部分包含新生代，新生代的垃圾收集仍然采用暂停所有应用线程的方式，将存活对象移动到老年代或者 Survivor 空间。同其他的收集算法一样，这些操作也利用多线程的方式完成。

G1 收集器属于 Concurrent 收集器：老年代的垃圾收集工作由后台线程完成，大多数的工作不需要暂停应用线程。由于老年代被划分到不同的区域，G1 收集器通过将对象从一个区域复制到另一个区域，完成对象的清理工作，这也意味着在正常的处理过程中，G1 收集器实现了堆的压缩整理（至少是部分的整理）。因此，使用 G1 收集器的堆不大容易发生碎片化——虽然这种问题无法避免。

同 CMS 收集器一样，避免 Full GC 的代价是消耗额外的 CPU 周期：负责垃圾收集的多个后台线程必须能在应用线程运行的同时获得足够的 CPU 运行周期。通过标志 `-XX:+UseG1GC`（默认值是关闭的）可以启动 G1 垃圾收集器。

触发及禁用显式的垃圾收集

通常情况下垃圾收集是由 JVM 在需要的时候触发：新生代用尽时会触发 Minor GC，老年代用尽时会触发 Full GC，或者堆空间即将填满时会触发 Concurrent 垃圾收集（如果情况需要）。

Java 也提供了一种机制让应用程序强制进行 GC：这就是 `System.gc()` 方法。通常情况下，试图通过调用这个方法显式触发 GC 都不是个好主意。调用这个方法会触发 Full GC（即使 JVM 使用 CMS 或者 G1 垃圾收集器），应用程序线程会因此而停顿相当长的一段时间。同时，调用这个方法也不会让应用程序更高效，它会让 GC 更早地开始，但那实际只是将性能的影响往后推迟而已。

任何原则都有例外，尤其是在做性能监控或者基准测试时。运行少量的代码进行基准测试时，为了更快地预热 JVM，在测量周期之前强制进行一次 GC 还是有意义的。类似的情况还包括在进行堆分析时，通常在获取堆转储之前，强制进行一次 Full GC 是一个不错的主意。虽然大多数抓取堆转储的方法都能进行 Full GC，也存在其他的方法可以强制进行 Full GC：你可以通过执行 **`jcmd`** <进程号> **`GC.run`**，或者使用 `jconsole` 连接到 JVM 在内存面板上单击“进行 GC”按钮。

另一个例外是 RMI，作为它分布式垃圾收集器的一部分，每隔一小时它会调用 `System.gc()` 一次。这里的调用时间可以通过系统属性中的 `-Dsun.rmi.dgc.server.gcInterval=`*`N`* 和 `-Dsun.rmi.dgc.cli ent.gcInterval=`*`N`* 进行修改。N 值的单位以毫秒记，在 Java 7（该值与之前的版本亦不相同）中的默认值为 3 600 000（即一个小时）。

如果你运行的程序调用的第三方代码中错误地调用了 `System.gc()` 方法，可以通过 JVM 参数 `-XX:+DisableExplicitGC` 显式地禁止这种类型的 GC；默认情况下该标志是关闭的。

快速小结

1. 这四种垃圾收集算法分别采用了的不同的方法来缓解 GC 对应用程序的影响。
2. Serial 收集器常用于仅有单 CPU 可用以及当其他程序会干扰 GC 的情况（通常是默认值）。
3. Throughput 收集器在其他的虚拟机上是默认值，它能最大化应用程序的总吞吐量，但是有些操作可能遭遇较长的停顿。

4. CMS 收集器能够在应用线程运行的同时并行地对老年代的垃圾进行收集。如果 CPU 的计算能力足以支撑后台垃圾收集线程的运行，该算法能避免应用程序发生 Full GC。
5. G1 收集器也能在应用线程运行的同时并发地对老年代的垃圾进行收集，在某种程度上能够减少发生 Full GC 的风险。G1 的设计理念使得它比 CMS 更不容易遭遇 Full GC。

5.1.3　选择GC算法

GC 算法的选择一方面取决于应用程序的特征，另一方面取决于应用的性能目标。

Serial 收集器最适用于应用程序的内存使用少于 100 MB 的场景。这种情况下应用程序只需要很小的堆，无论是 Throughput 收集器的并行收集，还是 CMS 收集器或 G1 收集器的后台收集都发挥不了太大的作用。

这个 Sizing 准则也限制了 Serial 收集器的使用范畴。大多数的程序需要在 Throughput 和 Concurrent 收集器之间做出抉择；而选择的依据大多数情况下是由应用程序的性能目标所决定的。

关于这个主题在第二章有过概述：不同应用在耗时、吞吐量、或者平均（或者总量 90% 的）响应时间上的要求迥异。

1. GC算法及批量任务

对批量任务而言，Throughput 收集器所引入的停顿，尤其是 Full GC 的停顿是主要的顾虑。每个任务的执行都会为总的执行时间增加一部分的延迟时间（elapse time）。如果每次 Full GC 耗时 0.5 秒，程序 5 分钟的运行时间内要进行 20 个这样的周期，那么性能的损耗就高达 3.4%：如果没有这些停顿，程序可以在 290 秒而不是 300 秒内完成运行。

如果有额外的 CPU 处理能力（这很可能是个问题），那么使用 Concurrent 收集器将极大地提升应用程序的性能。这里的关键在于我们能否提供足够的 CPU 给 Concurrent 收集器的线程进行后台的处理工作。举个简单的例子：一个单 CPU 的机器上，单线程的应用程序已经消耗了 100% 的 CPU 资源。该应用程序使用 Throughput 收集器运行时，GC 间歇性地发生，导致应用程序线程出现停顿。同样的程序，如果切换到 Concurrent 收集器，操作系统一会在 CPU 上运行应用程序线程，一会儿运行 GC 的后台线程。最终的结果是一样的：操作系统运行其他线程时，应用程序线程依然会发生停顿（不过有可能是更短时间的停顿）。

这个原则同样适用于通用情况，即多个应用程序线程、多个后台 GC 线程运行于多 CPU 的系统上。如果操作系统无法在同时运行所有应用程序线程和 GC 后台线程，那么对 CPU 的竞争就会反映到应用程序线程的停顿上。

表 5-1 展示了这个取舍是如何工作的。计算股票数据的批量应用已经运行于特定的模式，它们会将结果集保持在内存中数分钟（目的是填满整个堆）；测试分别使用了 CMS 和 Throughput 垃圾收集算法。

表5-1：采用不同的GC算法进行批量处理的时间消耗

垃圾收集算法	4核CPU（CPU利用率）	1核CPU（CPU利用率）
CMS	78.09（30.7%）	120.0（100%）
Throughput	81.00（27.7%）	111.6（100%）

表中的时间即是完成这个测试所花费的秒数，机器的 CPU 使用情况标记在括号中。存在 4 个可用 CPU 时，CMS 收集器运行批量操作比 Throughput 收集器要快大约 3 秒钟，但是请留意每个用例中 CPU 的使用情况。4 个 CPU 中有一个应用线程一直处于运行状态，因此应用线程使用了 4 个 CPU 的 25%。

表中其他额外的 CPU 消耗都源于 GC 线程。使用 CMS 收集器时，后台线程间歇性地消耗了一个 CPU，或者该机器上 25% 的 CPU 资源。这里后台线程是间歇性地运行的，结果表明，它消耗了大约 5% 的 CPU 时间，所以平均 CPU 使用率为 30%。

类似地，Throughput 收集器运行了 4 个 GC 线程。GC 周期中，这些线程占用了 100% 的可用 CPU 资源，在整个测试中，使用了大约 28% 的 CPU 周期。在 Minor GC 时，CMS 也运行了 4 个 GC 线程，占用了 100% 的可用 CPU。

只有一个 CPU 可用时，CPU 将一直处于忙碌状态，要么是运行应用程序线程，要么是运行 GC 线程。这种情况下，CMS 额外的后台线程就变成了一种负担，最终 Throughput 收集器提前 9 秒钟完成了运行任务。

平均 CPU 利用率和 GC

测试中如果只看平均 CPU 利用率可能会错过 GC 周期中的一些有趣的场景。Throughput 收集器运行时会（默认）100% 占用机器上所有的 CPU，因此测试中 CPU 使用情况更精准的表述应该如图 5-2 所示。

大多数情况下，如果只有应用程序线程运行，大概会消耗总 CPU 处理能力的 25%。一旦垃圾收集开始，CPU 就被 100% 占用。因此，即使测试中的平均值如图中水平虚线所示，实际的 CPU 使用情况更贴近于图中的锯齿形模式。

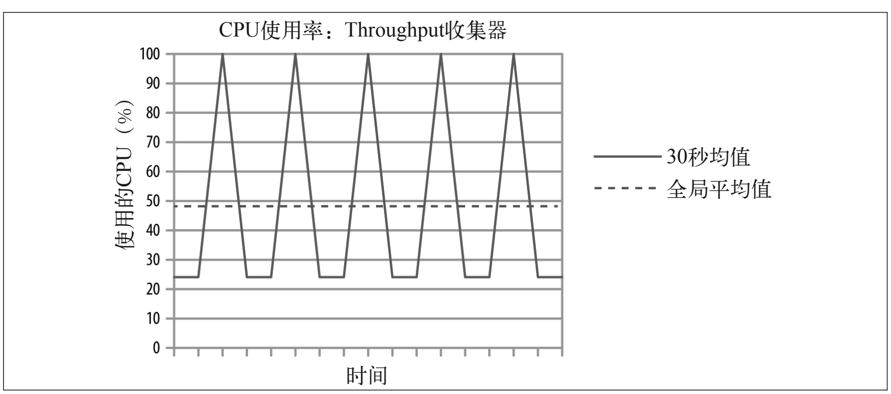


图 5-2：CPU 使用率的实际值与平均值（使用 Throughput 收集器）

使用 Concurrent 收集器时，后台线程会与应用线程并行运行，表现出的效果也不大一样。这种情况下 CPU 的使用情况看起来如图 5-3 所示。

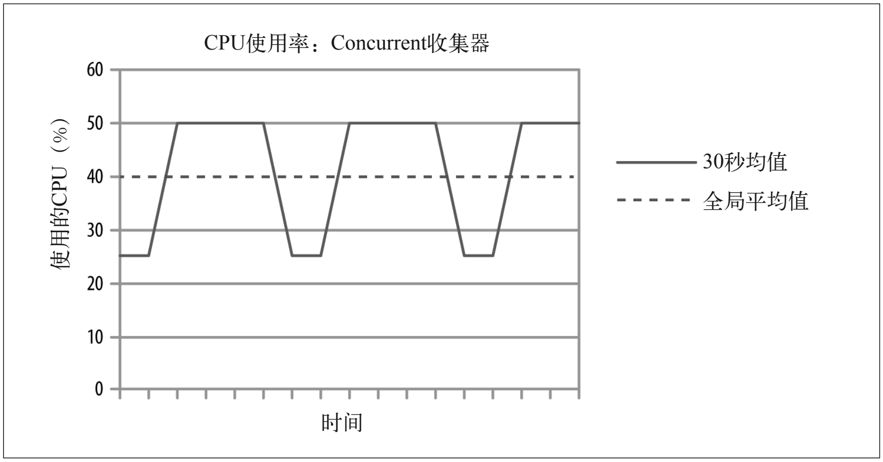


图 5-3：CPU 使用率的实际值与平均值（使用 CMS 收集器）

初始时应用线程使用了 25% 的 CPU 处理能力。到某个时刻线程产生足够的垃圾触发了 CMS 收集器的后台线程；垃圾回收的后台线程占用了另一颗 CPU，将 CPU 的使用率提高到 50%。CMS 线程的垃圾回收工作完成后，CPU 的使用率又回落到 25%，如此周而往复。注意，这个图中没有出现 CPU 使用 100% 的峰值时期，不过这是一种简化：CMS 新生代垃圾收集的过程中，实际可能有非常短的时段 CPU 使用率冲高到 100%，但是这些时段非常少，所以在这里的讨论中我们忽略了这部分时间。

Concurrent 收集器可以同时运行多个后台线程，其效果是类似的：这些后台线程运行时会消耗额外的 CPU 资源，进一步抬高了长期的 CPU 的平均使用率。

监控系统中定义的由 CPU 使用率触发的规则尤其重要：你需要确保 100% 的 CPU 使用率不是由 Full GC 所引起的临时性 CPU 暴涨，或者是由于后台并行处理线程所引起的持续时间更长（不过使用率稍低）的 CPU 高峰。在 Java 程序的世界里，这些峰值都是正常的状况。

快速小结

1. 使用 Throughput 收集器处理应用程序线程的批量任务能最大程度地利用 CPU 的处理能力，通常能获得更好的性能。
2. 如果批量任务并没有使用机器上所有可用的 CPU 资源，那么切换到 Concurrent 收集器往往能取得更好的性能。

2. GC算法和吞吐量测试

测试度量的目标是吞吐量时，选择 GC 算法的最基本的取舍跟批量任务一样，但是停顿所产生的影响却是大相径庭。CPU 仍然是影响总体性能非常重要的一环。

本节中，我们使用股票 servlet 程序作为测试的基准；servlet 程序运行在 GlashFish 服务器实例上，该 GlashFish 实例配有 2G 大小的堆，servlet 需要为每个用户的 HTTP 会话保存前 10 个 HTTP 请求（这个需求给系统的垃圾回收带来了更多的压力）。表 5-2 展示了分别使用 Throughput 收集器和 CMS 收集器，servlet 程序的吞吐量测试结果。测试运行的机器配置了 4 核的 CPU。

表5-2：使用不同的GC算法时吞吐量的情况

客户端会话数	Throughput TPS（CPU使用率）	CMS TPS（CPU使用率）
1	30.43（29%）	31.98（31%）
10	81.34（97%）	60.20（85%）

我们运行了两组测试来度量总的吞吐量。第一组测试使用了 `fhb` 程序来模拟单一的客户端；第二组测试使用了 10 个客户端发起负荷，最终目标机器的 CPU 被撑满。

存在空闲 CPU 周期时，CMS 收集器的性能更好，比 Throughput 收集器的每秒处理的事务数（TPS）高出大约 5%。Throughput 收集器在测试中经历了 24 次 Full GC 的停顿（停顿时它无法继续处理请求）；这些停顿约占测试稳定运行时间的 5%。通过避免这样的停顿，CMS 提供了更好的吞吐量。

然而，当 CPU 资源受限时，CMS 收集器的表现就差很多：比 Throughput 收集器的每秒吞吐量少大约 23.5%。注意，测试中，这时 CMS 收集器甚至无法让 CPU 以接近 100% 忙碌的程度运行。那是因为可用的 CPU 周期无法支撑后台的 CMS 收集线程运行，所以 CMS 收集器发生了并发模式失效（Concurrent Mode Failure）。发生这种失效意味着 JVM 不得不蜕化到单线程的 Full GC 模式，所以那段时间内（4 颗 CPU 的机器只有 25% 的 CPU 处于忙碌状态）平均 CPU 的使用率骤降。

3. GC算法及响应时间测试

表 5-3 表明请求之间的 Think Time 置为 250 毫秒时，使用同样的测试，负荷处理速度恒定在每秒 29 个事务。性能度量的标准是每个请求的平均响应时间，90% 的响应时间以及 99% 的响应时间。

表5-3：使用不同GC算法的响应时间

会话大小	Throughput收集器				CMS收集器			
	Avg	90th%	99th%	CPU	Avg	90th%	99th%	CPU
10 个元素	0.092	0.171	0.813	41%	0.104	0.211	0.260	46%
50 个元素	0.180	0.218	3.617	55%	0.107	0.222	0.315	53%

第一个测试在用户会话状态中保持了 10 个请求。对比这两种收集器，得到的结果是比较典型的：Throughput 收集器在平均响应时间甚至是 90% 响应时间的指标上都比 Concurrent 收集器快。但是 CMS 在 99% 响应时间上显示了巨大的优势：Throughput 收集器在完成剩下 1% 的垃圾收集工作上消耗了更长的时间（Full GC 期间这部分操作被完全停止了）。CMS 收集器用大约 10% 的 CPU 处理能力提升了 99% 时的响应时间的结果。

会话数据中的请求数增大到 50 个，GC 周期的影响就愈发明显，尤其是使用 Throughput 收

集器时。这时 Throughput 收集器的响应时间远远大于 CMS 收集器，这些都源于大量的超额负荷（outliers）将 99% 的响应时间不断拉长，有的甚至会超过 3 秒钟。不过还有一个有趣的现象，Throughput 收集器在 90% 响应时间的指标上低于 CMS 收集器——JVM 不进行 Full GC 的时候，Throughput 收集器依然持有优势。

这样的情况时不时会发生，但是其频率要远远低于第一种情况。某种程度上，上一个例子中，CMS 收集器是幸运的：通常是堆内保持了大量的活跃数据导致 Throughput 收集器进行 Full GC 的时间占用了大部分的响应时间，这种情况下使用 CMS 收集器也会发生并行模式失效（Concurrent Mode Failure）。而这个例子中，CMS 收集器的后台处理线程恰好能满足应用程序的需要。

以上这些是你在选择适合自身性能目标的垃圾回收算法时需要考虑的各种取舍。如果你关注的仅仅是平均响应时间，那么 Throughput 收集器和 Concurrent 收集器似乎差别不大，都能满足你的要求，你可以进一步考察 CPU 的使用情况（这时 Throughput 收集器可能是更优的选择）。如果你关注的是 90% 或者其他百分比的响应时间，那就只能通过性能测试来了解，完成这些任务应用程序会进行多少次 Full GC，最后决定选择哪种收集器。

快速小结

1. 衡量标准是响应时间或吞吐量，在 Throughput 收集器和 Concurrent 收集器之间做选择的依据主要是有多少空闲 CPU 资源能用于运行后台的并发线程。
2. 通常情况下，Throughput 收集器的平均响应时间比 Concurrent 收集器要差，但是在 90% 响应时间或者 99% 响应时间这几项指标上，Throughput 收集器比 Concurrent 收集器要好一些。
3. 使用 Throughput 收集器会超负荷地进行大量 Full GC 时，切换到 Concurrent 收集器通常能获得更低的响应时间。

4. CMS收集器和G1收集器之间的抉择

上一节的测试使用 CMS 收集器作为 Concurrent 收集器。一般情况下，堆空间小于 4 GB 时，CMS 收集器的性能比 G1 收集器好。CMS 收集器使用的算法比 G1 更简单，因此在比较简单的环境中（譬如堆的容量很小的情况），它运行得更快。使用大型堆或巨型堆时，由于 G1 收集器可以分割工作，通常它比 CMS 收集器表现更好。

回收任何对象之前，CMS 收集器的后台线程必须扫描完整个老年代空间。显然，扫描完整个堆的时间与堆的大小密切相关。如果堆还未填满之前，CMS 收集器的后台线程就停止了堆的扫描，直接回收对象，CMS 收集器会发生并发模式失效（Concurrent Mode Failure）：一旦发生这样的状况，CMS 收集器就不得不回退，暂停所有的应用线程，进行 Full GC 操作。这时处理 Full GC 的仅有唯一一个线程，性能的损耗非常严重。虽然通过调优 CMS 收集器，我们可以使用多个后台线程来减少变化带来的损失，不过随着堆的增大，CMS 后台线程需要处理的工作也越多。（CMS 收集器发生并发模式失效同时也会受应用程序的内存分配影响。）

G1 收集器采用了不同的方式来处理这个问题，它将老年代划分成不同的区域（Region），能更加容易地使用多个线程分担扫描老年代空间的任务。如果后台线程跟不上处理的速

度，G1 收集器也会发生并发模式失效，但是 G1 算法已经使得发生这种状况的几率减小了很多。

由于 CMS 收集器不对堆进行压缩整理（除非发生了耗时的 Full GC），堆的碎片化也会触发 CMS 收集器进行 Full GC。G1 算法在处理过程中随时进行着堆的压缩整理，不过 G1 收集器依然可能遭遇堆的碎片化问题，但是与 CMS 收集器比较起来，它的设计让它又领先了一步。

调优 CMS 收集器和 G1 收集器避免发生这些失效的方法很多，但对一些应用程序来说却不一定奏效。随着堆的不断增大（发生 Full GC 的代价变得更加昂贵），使用 G1 收集器更易于避免这些问题的发生。（另一方面，对有的程序，试图通过调优这两种收集器的任何一种避免发生并发模式失效几乎是不可能的任务。因此，即使应用程序的性能目标似乎与 Concurrent 收集器保持一致，使用 Throughput 收集器却可能是更明智的选择）。

最后，在这三种收集器的选择时还有一些微妙的无形因素需要考虑。Throughput 收集器是这三个收集器中年代最久远的一个，这意味着 JVM 工程师们已经花费了大量的时间精力雕琢把玩它，它的习性也更为大家所熟知。G1 作为相对较新的一种垃圾收集算法，更容易碰到设计时无法预期的极端情况。相对而言，G1 算法中影响性能的调优控制开关更少，这可能是好事，也可能是坏事。直到 Java 7u4，G1 都一直被当作实验版本，它的一些调优特性直到 Java 7u10 中才提供出来。相对于 Java 7 及之前的版本而言，G1 的性能提升主要体现在 Java 8 中。G1 将来的工作可能会关注在如何提高它在较小的堆上相对于 CMS 的性能优势。

快速小结

1. 选择 Concurrent 收集器时，如果堆较小，推荐使用 CMS 收集器。
2. G1 的设计使得它能够在不同的分区（Region）处理堆，因此它的扩展性更好，比 CMS 更易于处理超大堆的情况。

5.2 GC调优基础

虽然处理堆时各种 GC 算法有所差异，但是它们的基本配置参数是一致的。很多情况下，我们只需要这些基础的配置就能运行应用程序。

5.2.1 调整堆的大小

GC 调整的第一堂课是调整应用程序堆的大小。关于堆大小的调整还有更高级的话题，不过作为第一步，我们首先讨论如何设置总体堆的大小。

与其他的性能问题一样，选择堆的大小其实是一种平衡。如果分配的堆过于小，程序的大部分时间可能都消耗在 GC 上，没有足够的时间去运行应用程序的逻辑。但是，简单粗暴地设置一个特别大的堆也不是解决问题的方法。GC 停顿消耗的时间取决于堆的大小，如果增大堆的空间，停顿的持续时间也会变长。这种情况下，停顿的频率会变得更少，但是它们持续的时间会让程序的整体性能变慢。

使用超大堆还有另一个风险。操作系统使用虚拟内存机制管理机器的物理内存。一台机器

可能有 8 G 的物理内存，不过操作系统可能让你感觉有更多的可用内存。虚拟内存的数量取决于操作系统的设置，譬如操作系统可能让你感觉它的内存达到了 16 G。操作系统通过名为“交换”（swapping）（或者称之为分页，虽然这两者之间在技术上存在着差异，但是这些差异在这里不影响我们的讨论）。你可以载入需要 16 G 内存的应用程序，操作系统在需要时会将程序运行时不活跃的数据由内存复制到磁盘。再次需这部分内存的内容时，操作系统再将它们由磁盘重新载入到内存（为了腾出空间，通常它会先将另一部分内存的内容复制到磁盘）。

系统中运行着大量不同的应用程序时，这个流程工作得很顺畅，因为大多数的应用程序不会同时处于活跃状态。但是，对于 Java 应用，它工作得并不那么好。如果一个 Java 应用使用了这个系统上大约 12 G 的堆，操作系统可能在 RAM 上分配了 8 G 的堆空间，另外 4 G 的空间存在于磁盘（这个假设对实际情况进行了一些简化，因为应用程序也会使用部分的 RAM）。JVM 不会了解这些：操作系统完全屏蔽了内存交换的细节。这样，JVM 愉快地填满了分配给它的 12 G 堆空间。但这样就导致了严重的性能问题，因为操作系统需要将相当一部分的数据由磁盘交换到内存（这是一个昂贵操作的开始）。

更糟糕的是，这种原本期望一次性的内存交换操作在 Full GC 时一定会再次重演，因为 JVM 必须访问整个堆的内容。如果 Full GC 时系统发生内存交换，停顿时间会以正常停顿时间数个量级的方式增长。类似地，如果使用 Concurrent 收集器，后台线程在回收堆时，它的速度也可能会被拖慢，因为需要等待从磁盘复制数据到内存，结果导致发生代价昂贵的并发模式失效。

因此，调整堆大小时首要的原则就是永远不要将堆的容量设置得比机器的物理内存还大，另外，如果同一台机器上运行着多个 JVM 实例，这个原则适用于所有堆的总和。除此之外，你还需要为 JVM 自身以及机器上其他的应用程序预留一部分的内存空间：通常情况下，对于普通的操作系统，应该预留至少 1 G 的内存空间。

堆的大小由 2 个参数值控制：分别是初始值（通过 `-Xms` *N* 设置）和最大值（通过 `-Xmx` *N* 设置）。默认值的调节取决于多个因素，包括操作系统类型、系统内存大小、使用的 JVM。其他的命令行标志也会对该值造成影响；堆大小的调节是 JVM 自适应调优的核心。

JVM 的目标是依据系统可用的资源情况找到一个“合理的”默认初始值，当且仅当应用程序需要更多的内存（依据垃圾回收时消耗的时间来决定）时将堆的大小增大到一个合理的最大值。到目前为止，JVM 的高级调优标志以及调优细节都没有提及。为了让大家有一个感性的认识，我们列出了堆大小的默认最大值和最小值供大家参考，参见表 5-4。（为了使内存对齐，JVM 会对这些值进行圆整操作；所以 GC 日志中输出的大小可能与表中给出的值并不完全一致）。

表5-4：默认堆的大小

操作系统及JVM类型	初始堆的大小（Xms）	最大堆的大小（Xmx）
Linux/Solaris，32 位客户端	16 MB	256 MB
Linux/Soaris，32 位服务器	64 MB	取 1 GB 和物理内存大小 1/4 二者中的最小值
Linux/Soaris，64 位服务器	取 512 MB 和物理内存大小 1/64 二者中的最小值	取 32 GB 和物理内存大小 1/4 二者中的最小值

（续）

操作系统及JVM类型	初始堆的大小（Xms）	最大堆的大小（Xmx）
MacOS，64 位服务器型 JVM	64 MB	取 1 GB 和物理内存大小 1/4 二者中的最小值
32 位 Window 系统，客户端型 JVM	16 MB	256 MB
64 位 Window 系统，服务器型 JVM	64 MB	1 GB 和物理内存大小 1/4 二者中的最小值

如果机器的物理内存少于 192 MB，最大堆的大小会是物理内存的一半（大约 96 MB，或者更少）。

堆的大小具有初始值和最大值的这种设计让 JVM 能够根据实际的负荷情况更灵活地调整 JVM 的行为。如果 JVM 发现使用初始的堆大小，频繁地发生 GC，它就会尝试增大堆的空间，直到 JVM 的 GC 的频率回归到正常的范围，或者直到堆大小增大到它的上限值。

对很多应用来说，这意味着堆的大小不再需要调整了。实际上，你只需要为你选择的 GC 算法设定性能目标：譬如你能忍受的停顿持续时间、你期望垃圾回收在整个时间中所占用的百分比等。具体的细节设置取决于你选择的垃圾收集算法，在接下来的章节我们会进行深入的讨论（然而，即使到了那个时候，为了能尽可能地适用于更多的应用程序，减少调整的代价，仍然可能使用默认值）。

通常，如果应用程序运行需要的堆不会使用超过运行平台默认的最大值，这个方法就工作得非常好。然而，如果应用程序在 GC 时消耗了太长的时间，你很有可能需要使用 `-Xmx` 标志增大堆的大小。选择什么样的大小没有一个硬性的或简单的规则（不过你需要确保设置的大小是机器可以支持的）。一个经验法则是完成 Full GC 后，应该释放出 70% 的空间（30% 的空间仍然占用）。为了衡量这个结果，你可以持续运行应用程序，直到其到达稳定态配置：这时它已经载入了需要缓存的所有对象，或者已经创建了最多的客户端连接数，诸如此类。之后，使用 `jconsole` 连接应用程序，强制进行 Full GC，观察 Full GC 结束后还有多少内存被占用（此外，对于 Throughput 垃圾收集器，如果有日志的话，你可以通过查询 GC 日志得到对应的数据）。

注意，即使你显式地设置了堆的最大容量，还是会发生堆的自动调节：初始时堆以默认的大小开始运行，为了达到根据垃圾收集算法设置的性能目标，JVM 会逐步增大堆的大小。将堆的大小设置得比实际需要更大不一定会带来性能损耗：堆并不会无限地增大，JVM 会调节堆的大小直到其满足 GC 的性能目标。

另一方面，如果你确切地了解应用程序需要多大的堆，那么你可以将堆的初始值和最大值直接设置成对应的数值（譬如：`-Xms4096m -Xmx4096m`）。这种设置能稍微提高 GC 的运行效率，因为它不再需要估算堆是否需要调整大小了。

快速小结

1. JVM 会根据其运行的机器，尝试估算合适的最大、最小堆的大小。
2. 除非应用程序需要比默认值更大的堆，否则在进行调优时，尽量考虑通过调整 GC 算法的性能目标（具体内容在下一章介绍），而非微调堆的大小来改善程序性能。

5.2.2 代空间的调整

一旦堆的大小确定下来，你（或者 JVM）就需要决定分配多少堆给新生代空间，多少给老年代空间。我们应该清楚地了解代的划分对性能的影响：如果新生代分配得比较大，垃圾收集发生的频率就比较低，从新生代晋升到老年代的对象也更少。任何事物都有两面性，采用这种分配方法，老年代就相对比较小，比较容易被填满，会更频繁地触发 Full GC。这里找到一个恰当的平衡点是解决问题的关键。

不同的 GC 算法尝试使用不同的方法来解决这些平衡问题。虽然方法不同，不过所有的 GC 方法都使用了同一套标志来设置代的大小；这一节会详细介绍这些通用的标志。

所有用于调整代空间的命令行标志调整的都是新生代空间；新生代空间剩下的所有空间都被老年代占用。多个标志都能用于新生代空间的调整，它们分别如下所列。

`-XX:NewRatio=`*`N`*

设置新生代与老年代的空间占用比率。

`-XX:NewSize=`*`N`*

设置新生代空间的初始大小。

`-XX:MaxNewSize=`*`N`*

设置新生代空间的最大大小。

`-Xmn`*`N`*

将 `NewSize` 和 `MaxNewSize` 设定为同一个值的快捷方法。

最初新生代空间大小是由 `NewRatio` 指定大小，`NewRatio` 的默认值为 2。影响堆空间大小的参数通常以比率的方式指定；这个值被用于一个计算空间分配的公式之中。下面是使用 `NewRatio` 计算空间的公式：

```
Initial Young Gen Size = Initial Heap Size / (1 + NewRatio)
```

代入堆的初始大小和 `NewRatio` 的值就能得到新生代的设置值。那么我们很容易得出，默认情况下，新生代空间的大小是初始堆大小的 33%。

除此之外，新生代的大小也可以通过 `NewSize` 标志显式地设定。使用 `NewSize` 标志设定的新生代大小，其优先级要高于通过 `NewRatio` 计算出来的新生代大小。`NewSize` 标志没有默认的设置（虽然使用 `Printflagsfinal` 标志输出的值为 1 MB）。`NewSize` 不设置的情况下，初始的新生代大小由 `NewRatio` 计算出的值决定。

如果堆的大小扩张，新生代的大小也会随之增大，直到由 `MaxNewSize` 标志设定的最大容量。默认情况下，新生代的最大值也是由 `NewRatio` 的值设定的，不过它也同时受制于堆的最大容量（注意，不是初始大小）。

试图通过指定新生代的最大及最小值区间的方式调优新生代的结果是十分困难的。如果堆的大小是固定的（可以通过将 `-Xms` 和 `-Xmx` 指定为相等的值实现），通常推荐使用 `-Xmn` 标志将新生代也设定为固定大小。如果应用程序需要动态调整堆的大小，并希望有一个更大（或者更小）的新生代，那就需要关注 `NewRatio` 值的设定。

快速小结

1. 整个堆范围内，不同代的大小划分是由新生代所占用的空间控制的。
2. 新生代的大小会随着整个堆大小的增大而增长，但这也是随着整个堆的空间比率波动变化的（依据新生代的初始值和最大值）。

5.2.3　永久代和元空间的调整

JVM 载入类的时候，它需要记录这些类的元数据。从终端用户的角度来看，这些只是一些“书签”信息。这部分数据被保存在一个单独的堆空间中。在 Java 7 里，这部分空间被称为永久代（Permgen 或 Permanent Generation），在 Java 8 中，它们被称为元空间（Metaspace）。

不过永久代和元空间并不完全一样。Java 7 中，永久代还保存了一些与类数据无关的杂项对象（miscellaneous object）；这些对象在 Java 8 中被挪到了普通的堆空间内。除此之外，Java 8 还从根本上改变了保存在这个特殊区域内的元数据的类型——不过由于普通用户不需要了解这个区域内保持了什么样的数据，所以这些改变不会对我们造成什么影响。作为终端用户，我们需要知道的仅仅是永久代级元空间内保存了大量与类相关的数据，有些时候我们可能会需要调整这部分空间的大小。

注意永久代或者元空间内并没有保存类实例的具体信息（即类对象），也没有反射对象（譬如方法对象）；这些内容都保存在常规的堆空间内。永久代和元空间内保存的信息只对编译器或者 JVM 的运行时有用，这部分信息被称为“类的元数据”。

到目前为止都没有一个能提前计算出程序的永久代 / 元空间需要多大空间的好算法。永久代或者元空间的大小与程序使用的类的数量成比率相关，应用程序越复杂，使用的对象越多，永久代或者元空间就越大。使用元空间替换掉永久代的优势之一是我们不再需要对其进行调整——因为（不像永久代）元空间默认使用尽可能多的空间。表 5-5 列出了永久代和元空间的初始值及最大值。

表5-5：永久代/元空间的默认大小

JVM类型	默认的初始大小	默认永久代大小的最大值	默认元空间大小的最大值
32 位客户端型 JVM	12 MB	64 MB	没有限制
32 位服务器型 JVM	16 MB	64 MB	没有限制
64 位 JVM	20.75 MB	82 MB	没有限制

这些内存区域的行为就像是分隔开的普通堆空间。它们会根据初始的大小动态地调整，需要的时候会增大到最大的堆空间。对于永久代而言，可以通过 `-XX:PermSize=N`、`-XX:MaxPermSize=N` 标志调整大小。而元空间的大小可以通过 `-XX:MetaspaceSize=N` 和 `-XX:MaxMetaspaceSize=N` 调整。

> **元空间会过大吗？**
>
> 由于元空间默认的大小是没有作限制的，因此 Java 8（尤其是 32 位系统）的应用可能由于元空间被填满而耗尽内存。第 8 章中介绍的工具本地内存跟踪器（Native Memory Tracking，NMT）可以帮助诊断这种类型的问题。如果元空间增长得过大，通过设置 `MaxMetaspaceSize` 你可以调整元空间的上限，将其限制为一个更小的值，不过这又会导致应用程序最后由于元空间耗尽，发生 `OutOfMemoryError` 异常。解决这类问题的终极方法还是定位出为什么类的元空间会变得如此巨大。

调整这些区间会触发 Full GC，所以是一种代价昂贵的操作。如果程序在启动时发生大量的 Full GC（因为需要载入数量巨大的类），通常都是由于永久代或者元空间发生了大小调整，因此这种情况下为了改善启动速度，增大初始值是个不错的主意。对于定义了大量类的 Java 7 应用，同时还需要增大永久代空间的最大值。譬如，通常情况下应用服务器永久代的最大值会设置为 128 MB、192 MB 或者更多。

虽然名称叫“永久代”，保存在永久代空间中的数据并不能永久保存（元空间这个名字可能更准确）。尤其是，保存在其中的类像其他的对象一样会经历垃圾回收。在应用服务器中，这是一种非常普遍的现象，每次有新的应用部署，应用服务器都会创建新的类加载器（classloader）。之后老的类加载器就不再被引用，像它定义的任何一个类一样，等待 GC 的回收。应用服务器漫长的运行周期中，很容易发现部署中触发的 Full GC：永久代或元空间被新的类所充斥填满，老的类的元数据等待被回收。

堆转储（参见第 7 章）的信息可以用于诊断存在哪些类加载器，而这些信息反过来可以帮助确定是否存在类加载器的泄漏，最终导致永久代（或者元空间）被耗尽。除此之外，使用 `jmap` 和 `-permstat` 参数（适用于 Java 7）、或者 `-clstats` 参数（适用于 Java 8）可以输出类加载器相关的信息。不过这些命令都不是非常稳定，所以不大推荐使用。

快速小结

1. 永久代或元空间保存着类的元数据（并非类本体的数据）。它以分离的堆的形式存在。
2. 典型应用程序在启动后不需要载入新的类，这个区域的初始值可以依据所有类都加载后的情况设置。使用优化的初始值能够加速启动的过程。
3. 开发中的应用服务器（或者任何需要频繁重新载入类的环境）上经常能碰到由于永久代或元空间耗尽触发的 Full GC，这时老的元数据会被丢弃回收。

5.2.4　控制并发

除 Serial 收集器之外几乎所有的垃圾收集器使用的算法都基于多线程。启动的线程数由 `-XX:ParallelGCThreads=N` 参数控制。对下面这些操作，这个参数值会影响线程的数目：

- 使用 `-XX:+UseParallelGC` 收集新生代空间
- 使用 `-XX:+UseParallelOldGC` 收集老年代空间

- 使用 `-XX:+UseParNewGC` 收集新生代空间
- 使用 `-XX:+UseG1GC` 收集新生代空间
- CMS 收集器的“时空停顿”阶段（但并非 Full GC）
- G1 收集器的“时空停顿”阶段（但并非 Full GC）

由于 GC 操作会暂停所有的应用程序线程，JVM 为了尽量缩短停顿时间就必须尽可能地利用更多的 CPU 资源。这意味着，默认情况下，JVM 会在机器的每个 CPU 上运行一个线程，最多同时运行 8 个。一旦达到这个上限，JVM 会调整算法，每超出 5/8 个 CPU 启动一个新的线程。所以总的线程数就是（这里的 N 代表 CPU 的数目）：

```
ParallelGCThreads = 8 + ((N - 8) * 5 / 8)
```

有时候使用这个算法估算出来的线程数目会偏大。如果应用程序使用一个较小的堆（譬如大小为 1 GB）运行在一个八颗 CPU 的机器上，使用 4 个线程或者 6 个线程处理这个堆可能会更高效。在一个 128 颗 CPU 的机器上，启动 83 个垃圾收集线程可能也太多了，除非系统使用的堆已经达到了最大上限。

除此之外，如果机器上同时运行了多个 JVM 实例，限制所有 JVM 使用的线程总数是个不错的主意。这时，垃圾收集线程运行起来会更加高效，每个线程都能 100% 地利用各 CPU 的资源（这就是前面的例子中 Throughput 收集器的平均 CPU 使用率比预期值更高的原因）。在 8 核或者 CPU 更少的机器上，垃圾收集线程会 100% 地占用机器的 CPU 处理资源。在拥有更多 CPU、运行了多个 JVM 的机器上，通常出现的问题是有太多的垃圾回收线程在同时并发运行。

以 16 核 CPU 的机器同时运行 4 个 JVM 实例为例，每个 JVM 默认会启动 13 个垃圾收集线程。如果四个 JVM 同时进行垃圾回收操作，机器上会启动大约 52 个 CPU 密集型线程竞争 CPU 资源。这会导致大量的冲突，如果能够限制每个 JVM 最多启动 4 个垃圾收集线程，效率会高很多。即使在同一个时刻，4 个 JVM 中的线程不大可能同时进行 GC 操作，一个 JVM 上同时运行 13 个线程也意味着其他 JVM 上的应用程序线程不得不在一台总共有 16 个 CPU，且其中 13 个 CPU 被繁忙的垃圾收集任务 100% 占用的机器上竞争资源。这种情况下，将每个 JVM 的垃圾收集线程数限制到 4 个是一个比较合理的平衡。注意，这个标志不会对 CMS 收集器或者 G1 收集器的后台线程数作设定（虽然它们也会受设置的影响）。关于其中的细节，我们会在下一章中介绍。

快速小结

1. 几乎所有的垃圾收集算法中基本的垃圾回收线程数都依据机器上的 CPU 数目计算得出。
2. 多个 JVM 运行于同一台物理机上时，依据公式计算出的线程数可能过高，必须进行优化（减少）。

5.2.5 自适应调整

根据调优的策略，JVM 会不断地尝试，寻找优化性能的机会，所以在 JVM 的运行过程中，堆、代以及 Survivor 空间的大小都可能会发生变化。

这是一种尽力而为（Best-Effort）的方案，它进行性能调优的依据是以往的性能历史：这其中隐含了一个假设，即将来 GC 周期的状况跟最近历史 GC 周期的状况可能很类似。事实证明，在多种负荷下这一假设都是合理的，即使某个时刻内存的分配发生突变的情况，JVM 也能够依据最新的情况重新调整它的大小。

自适应调整在两个方面能提供重要的帮助。其一，这意味着小型应用程序不需要再为指定了过大的堆而担心。譬如用于调整应用服务器的命令行管理程序，这类型的程序通常使用 16 MB（或者 64 MB）的堆，即使默认的堆可能增长到 1 GB 那么大的容量。有了自适应调整之后，这种类型的应用程序不再需要额外花费精力去调优，平台默认的配置就能确保他们不会使用大量的内存。

其次，这意味着很多应用程序根本不需要担心它们堆的大小，如果需要使用的堆的大小超过了平台的默认值，他们可以放心地分配更大的堆，而不用关心其他的细节。JVM 会自动调整堆和代的大小，依据垃圾回收算法的性能目标，使用优化的内存量。自适应调整就是让自动调整生效的法宝。

不过，空间大小的调整终归要花费一定的时间开销，这部分时间大多数消耗在 GC 停顿的时候。如果你投注了大量的时间精细地调优了垃圾回收的参数、定义了应用程序堆的大小限制，可以考虑关闭自适应调整。如果应用程序的运行明显地可以划分成不同的阶段，你希望对这些阶段中的某一个阶段进行垃圾回收的优化，那么关闭自适应调优也是很有帮助的。

使用 `-XX:-UseAdaptiveSizePolicy` 标志可以在全局范围内关闭自适应调整功能（默认情况下，这个标志是开启的）。如果堆容量的最大、最小值设置成同样的值，与此同时新生代的初始值和最大值也设置为同样大小，自适应调整的功能会被关闭。不过这时的 Survivor 空间是个例外，我们在下一章中会详细介绍其中的细节。

如果你想了解应用程序运行时 JVM 的空间是如何调整的，可以设置 `-XX:+PrintAdaptiveSizePolicy` 标志。开启该标志后，一旦发生垃圾回收，GC 的日志中会包含垃圾回收时不同的代进行空间调整的细节信息。

快速小结

1. JVM 在堆的内部如何调整新生代及老年代的百分比是由自适应调整机制控制的。
2. 通常情况下，我们应该开启自适应调整，因为垃圾回收算法依赖于调整后的代的大小来达到它停顿时间的性能目标。
3. 对于已经精细调优过的堆，关闭自适应调整能获得一定的性能提升。

5.3 垃圾回收工具

由于垃圾回收对 Java 的性能影响至关重要，业界提供了大量的工具用于监控它的性能。

观察垃圾回收对应用程序的性能影响最好的方法就是尽量熟悉垃圾回收的日志，垃圾回收日志中包含了程序运行过程中的每一次垃圾回收操作。

垃圾回收日志的细节依据使用的垃圾回收算法各有不同，不过垃圾回收日志的基本结构（management）是一致的。这一节我们会介绍这些结构，与具体垃圾回收算法相关的更多日志细节会在下一章中介绍。

多种方法都能开启 GC 的日志功能，其中包括：使用 `-verbose:gc` 或 `-XX:+PrintGC` 这两个标志中的任意一个能创建基本的 GC 日志（这两个日志标志实际上互为别名，默认情况下的 GC 日志功能是关闭的）。使用 `-XX:+PrintGCDetails` 标志会创建更详细的 GC 日志。我们推荐使用 `-XX:+PrintGCDetails` 标志（这个标志默认情况下也是关闭的）；通常情况下使用基本的 GC 日志很难诊断垃圾回收时发生的问题。除了使用详细的 GC 日志，我们还推荐使用 `-XX:+PrintGCTimeStamps` 或者 `-XX:+PrintGCDateStamps`，便于我们更精确地判断几次 GC 操作之间的时间。这两个参数之间的差别在于时间戳是相对于 0（依据 JVM 启动的时间）的值，而日期戳（date stamp）是实际的日期字符串。由于日期戳需要进行格式化，所以它的效率可能会受轻微的影响，不过这种操作并不频繁，它造成的影响也很难被我们感知。

默认情况下 GC 日志直接输出到标准输出，不过使用 `-Xloggc:filename` 标志也能修改输出到某个文件。除非显式地使用 `-PrintGCDetails` 标志，否则使用 `-Xloggc` 会自动地开启基本日志模式。使用日志循环（Log rotation）标志可以限制保存在 GC 日志中的数据量；对于需要长时间运行的服务器而言，这是一个非常有用的标志，否则累积几个月的数据很可能会耗尽服务器的磁盘。通过 `-XX:+UseGCLogfileRotation -XX:NumberOfGCLogfiles=N -XX:GCLogfileSize=N` 标志可以控制日志文件的循环。默认情况下，`UseGCLogfileRotation` 标志是关闭的。开启 `UseGCLogfileRotation` 标志后，默认的文件数目是 0（意味着不作任何限制），默认的日志文件大小是 0（同样也是不作任何限制）。因此，为了让日志循环功能真正生效，我们必须为所有这些标志设定值。需要注意的是，如果设定的数值不足 8 KB 的话，日志文件的大小会以 8 KB 为单位规整。

根据需要，你可以手工地解析、研读垃圾回收日志，也可以利用一些工具来完成这部分工作。GC Histogram（http://java.net/projects/gchisto）就是这些工具中的一员。GC Histogram 能够读入 GC 日志，根据日志文件中的数据生成对应的图表和表格。图 5-4 是由 GC Histogram 生成的 GC 开销的概略表。

	Num	Num (%)	Total GC (s...	Total GC (%)	Overhead (...	Avg (ms)	Sigma (ms)	Min (ms)	Max (ms)
All	474	100.00%	193.246	100.00%	41.29%	407.693	1,347.194	57.254	9,592.072
Young GC	457	96.41%	71.705	37.11%	15.32%	156.904	44.162	57.254	260.142
Full GC	17	3.59%	121.541	62.89%	25.97%	7,149.478	1,873.358	224.791	9,592.072

图 5-4：GC Histogram 的停顿状态选项卡

在这个例子中，JVM 消耗了 41% 的时间进行垃圾回收，完成一次 Full GC 的时间长达 7 秒钟。很明显，这个应用程序需要调优它的内存使用。

使用 `jconsole` 可以实时监控堆的使用情况。`jconsole` 的内存面板可以实时查看堆的使用状况，如图 5-5 所示。

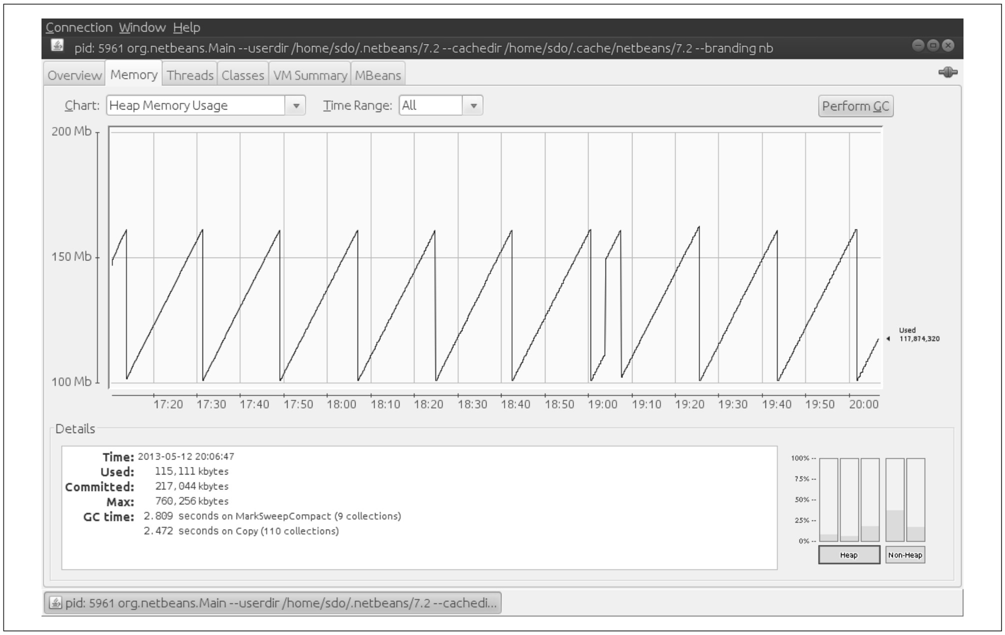


图 5-5：查看堆的实时状况

从这幅视图我们能看到整个堆的使用情况，它在介于 100 MB 到 160 MB 的区间内周期性地循环。使用 `jconsole` 一次只能看到一个分区的使用情况：要么是 Eden 空间，要么是 Survivor 空间，要么是老年代，或者是永久代。如果选择 Eden 空间作为绘制图表的区域，能看到 Eden 空间以相似的模式在 0 MB 到 60 MB 之间波动（并且，跟你的猜测一样，如果依据老年代的数据作图，那它基本将是一条横在 100 MB 的水平线）。

如果你希望使用脚本的方式获取数据，`jstat` 是理想的工具。`jstat` 提供了 9 个选项，提供堆的各种数据；使用 `jstat -options` 选项能够列出所有这些选项。这其中最常用的一个选项是 `-gcutil`，它能够输出消耗在 GC 上的时间，以及每个 GC 区域使用的百分比。其他的选项能够以 KB 为单位输出各 GC 空间的大小。

注意，`jstat` 接受一个可选的参数，指定每隔多少毫秒重复执行这个命令，这个选项帮助我们长时间地监控应用程序的垃圾回收过程。下面是一个示例的输出，它以每隔一秒钟的频率运行。

```
% jstat -gcutil process_id 1000
  S0     S1     E      O      P     YGC     YGCT    FGC    FGCT     GCT
 51.71   0.00  99.12  60.00  99.93     98    1.985     8    2.397   4.382
  0.00  42.08   5.55  60.98  99.93     99    2.016     8    2.397   4.413
  0.00  42.08   6.32  60.98  99.93     99    2.016     8    2.397   4.413
  0.00  42.08  68.06  60.98  99.93     99    2.016     8    2.397   4.413
  0.00  42.08  82.27  60.98  99.93     99    2.016     8    2.397   4.413
  0.00  42.08  96.67  60.98  99.93     99    2.016     8    2.397   4.413
  0.00  42.08  99.30  60.98  99.93     99    2.016     8    2.397   4.413
 44.54   0.00   1.38  60.98  99.93    100    2.042     8    2.397   4.439
 44.54   0.00   1.91  60.98  99.93    100    2.042     8    2.397   4.439
```

在这个例子中，监控开始时，程序已经在新生代（`YGC`）中进行了 99 次垃圾回收操作，这总共消耗了大约 1.985 秒的时间（`YGCT`）。于此同时，它还完成了 8 次 Full GC（`FGC`），消耗了 2.397 秒的时间（`FGCT`）；因此 GC 消耗的总时长（`GCT`）为 4.382 秒。

新生代中三个区间的数据都在这里列出：两个 Survivor 空间（分别是 `S0` 和 `S1`）以及 1 个 Eden 空间（标记为 `E`）。监控开始时，Eden 空间几乎要被填满了（已经占用了 99.12% 的空间），因此下一秒就有一次新生代的垃圾回收：这之后 Eden 空间的使用率回落到 5.55%，Survivor 空间发生了交换，一部分内存对象被晋升到了老年代空间（标记为 `O`），老年代的空间使用率增长到 60.98%。跟典型的场景一样，我们没有在永久代（标记为 `P`）发现大幅度的变化，因为几乎所有需要的类都已经在程序启动时载入内存。

如果你不记得如何开启 GC 日志，这是一个很好的替代方法，它能帮助我们观察垃圾回收是如何在较长的时间跨度内工作的。

快速小结

1. GC 日志是分析 GC 相关问题的重要线索；我们应该开启 GC 日志标志（即使是在生产服务器上）。
2. 使用 `PrintGCDetails` 标志能获得更详尽的 GC 日志信息。
3. 使用工具能很有效地帮助我们解析和理解 GC 日志的内容，尤其是在对 GC 日志中的数据进行归纳汇总时，它们非常有帮助。
4. 使用 `jstat` 能动态地观察运行程序的垃圾回收操作。

5.4 小结

对任何一个 Java 应用程序而言，垃圾收集的性能都是其构成整体性能的关键一环。虽然对大多数的应用程序来说，调优的工作仅仅是选择合适的垃圾收集算法，或者在需要的时候，增大应用程序的堆空间。

自适应调整让 JVM 能够自动地调整它的行为，使用给定的堆，提供尽可能好的性能。

更复杂的应用往往需要额外的调优，尤其是针对特定 GC 算法的调优。如果这一章中简单的 GC 设置并没有解决应用程序的性能问题，请参考下一章中介绍的调优内容。

第 6 章

垃圾收集算法

第 5 章介绍了垃圾收集器的通用行为，包括普遍适用于所有垃圾回收算法的 JVM 调优标志：如何选择堆的大小，如何选择代的大小，如何开启和设置 GC 日志，等等。

这些基础的调优标志已经足以应付大多数的场景。当它们无法解决问题时，往往需要查看使用的 GC 算法中具体是哪些操作影响了性能，进一步判断如何调整对应的参数，从而最大程度地减少 GC 操作对应用程序性能的影响。

调优特定收集器最要紧的信息是启动垃圾收集器后 GC 日志中的数据。从本章开始，我们会从 GC 的日志输出角度详细分析每种垃圾收集算法的行为；分析 GC 日志能帮助我们更好地理解垃圾收集算法是如何工作的，以及怎样调节参数能让它们工作得更好。之后的每一节都会包含一个性能调优的实战例子。

还有一些其他的因素也会影响几乎所有垃圾回收算法的性能，包括分配巨型对象、对象的生命周期既不长又不短，等等。我们会在本章的末尾讨论这些场景。

6.1 理解Throughput收集器

我们会逐一分析各个垃圾收集器的行为，首先会介绍 Throughput 收集器。Throughput 收集器有两个基本的操作：其一是回收新生代的垃圾，其二是回收老年代的垃圾。

图 6-1 展示了堆在新生代回收之前和回收之后的情况。

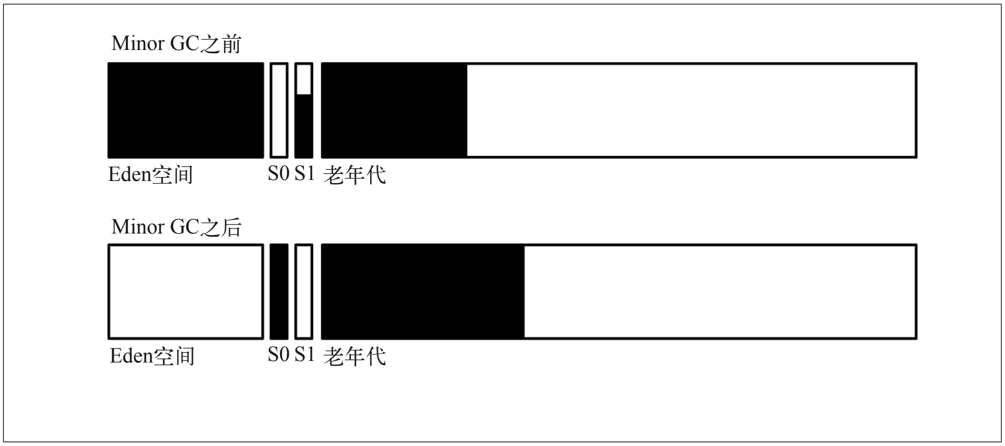


图 6-1：Throughput 垃圾回收中的新生代

通常新生代的垃圾回收发生在 Eden 空间快用尽时。新生代垃圾收集会把 Eden 空间中的所有对象挪走：一部分对象会被移动到 Survivor 空间（即这幅图中的 S0 区域），其他的会被移动到老年代；正如你看到的，回收之后老年代中保存了更多的对象。当然，还有大量的对象因为没有任何对象引用而被回收。

开启了 `PrintGCDetails` 标志的 GC 日志中，Minor GC 形式如下：

```
17.806: [GC [PSYoungGen: 227983K->14463K(264128K)]
            280122K->66610K(613696K), 0.0169320 secs]
            [Times: user=0.05 sys=0.00, real=0.02 secs]
```

这次 GC 在程序开始运行 17.806 秒后发生。现在新生代中对象占用的空间为 14 463 KB（约为 14 MB，位于 Survivor 空间内）；GC 之前，新生代对象占用的空间为 227 983 KB（约为 227 MB）。（实际上，227 893 KB 严格折算只有 222 MB，为了便于讨论，本章中以 1000 为单位将它们折算到 KB。这里假设我是磁盘生产商。）新生代这时总的大小为 264 MB。

与此同时，堆的空间总的使用情况（包含新生代和老年代）从 280 MB 减少到了 66 MB，这个时刻整个堆的大小为 613 MB。完成垃圾回收操作耗时 0.02 秒（排在输出最后的 Real 时间是 0.0 169 320 秒——实际时间进行了归整）。程序消耗的 CPU 时间比 Real 时间往往更多，原因是新生代垃圾回收会使用多个线程（这个例子中，使用了 4 个线程）。

图 6-2 展示了 Full GC 之前及之后堆的使用情况。

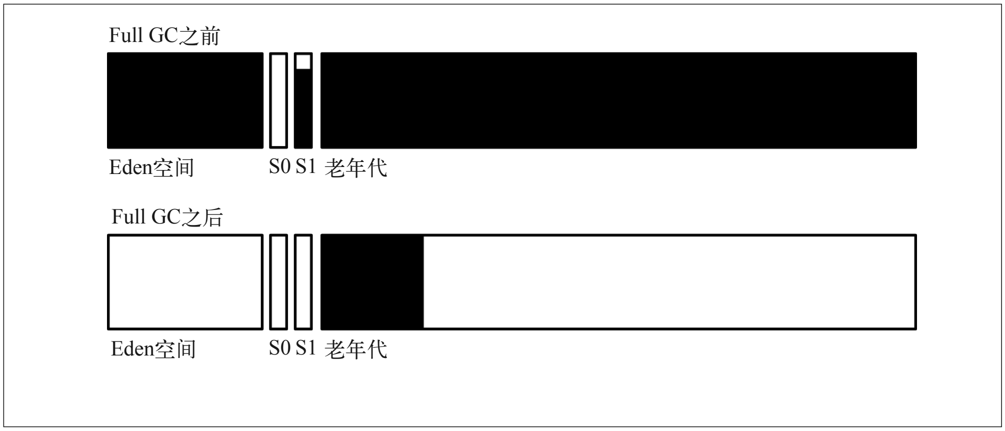


图 6-2：使用 Throughput 收集器的 Full GC

老年代垃圾收集会回收新生代中所有的对象（包括 Survivor 空间中的对象）。只有那些有活跃引用的对象，或者已经经过压缩整理的对象（它们占据了老年代的开始部分）会在老年代中继续保持，其余的对象都会被回收。

Full GC 的日志输出示例如下：

```
64.546: [Full GC [PSYoungGen: 15808K->0K(339456K)]
          [ParOldGen: 457753K->392528K(554432K)] 473561K->392528K(893888K)
          [PSPermGen: 56728K->56728K(115392K)], 1.3367080 secs]
          [Times: user=4.44 sys=0.01, real=1.34 secs]
```

新生代的空间使用在经历 Full GC 之后变为 0 字节（新生代的大小为 339 MB）。老年代中的空间使用从 457 MB 减少到了 392 MB，因此整个堆的使用从 473 MB 减少到了 392 MB。永久代空间的使用没有发生变化；在多数的 Full GC 中，永久代的对象都不会被回收。（如果永久代空间耗尽，JVM 会发起 Full GC 回收永久代中的对象，这时你会观察到永久代空间的变化——这是永久代进行回收唯一的情况。这个例子使用的是 Java 7；在 Java 8 中，类似的信息可以在元空间中找到）。由于 Full GC 要进行大量的工作，所以消耗了约 1.3 秒的 Real 时间，4.4 秒的 CPU 时间（同样源于使用了 4 个并行的线程）。



快速小结

1. Throughput 收集器会进行两种操作，分别是 Minor GC 和 Full GC。
2. 通过 GC 日志中的时间输出，我们可以迅速地判断出 Throughput 收集器的 GC 操作对应用程序总体性能的影响。

堆大小的自适应调整和静态调整

Throughput 收集器的调优几乎都是围绕停顿时间进行的，寻求堆的总体大小、新生代的大小以及老年代大小之间平衡。

考虑 Throughput 收集器的调优方案时有两种取舍。首先比较经典的是编程技术上的取舍，即时间与空间的取舍。

第二个取舍与完成垃圾回收所需的时长相关。增大堆能够减少 Full GC 停顿发生的频率，但也有其局限性：由于 GC 时间变得更长，平均响应时间也会变长。类似地，为新生代分配更多的堆空间可以缩短 Full GC 的停顿时间，不过这又会增大老年代垃圾回收的频率（因为老年代空间保持不变或者变得更小了）。

图 6-3 展示了采用这些取舍的效果。图上显示的是运行在 GlassFish 实例上的股票 Servlet 应用，在使用不同大小的堆时，最大吞吐量的变化情况。使用 256 MB 的小堆时，应用服务器在垃圾回收上消耗了大量的时间（实际消耗的时间高达总时间的 36%）；吞吐量因此受到限制，比较低。随着堆大小的增加，吞吐量迅速提升——直到堆的容量增大到 1500 MB。这之后吞吐量的增速迅速减缓，这时应用程序实际已经不太受垃圾回收的影响（垃圾回收消耗的时间仅仅只占总时间的 6% 左右）。收益递减规律逐渐凸显出来：虽然应用程序可以通过增加内存的方式提升吞吐量，不过其效果已经很有限了。

堆的大小达到 4500 MB 后，吞吐量开始出现少量下滑。这时，应用程序面临着第二个选择：增加的内存导致 GC 周期愈加冗长，虽然它们发生的频率小得多，但这些超长的 GC 周期也会影响系统整体的吞吐量。

这幅图中的数据取自关闭了自适应调整的 JVM；它的最大、最小堆的容量设置成了同样的大小。对任何一种应用，我们都可以通过实验确定堆和代的最佳大小，但是，让 JVM 自己来选择通常是更容易的方法（这也是最通常的做法，因为默认情况下自适应调整就是开启的）。

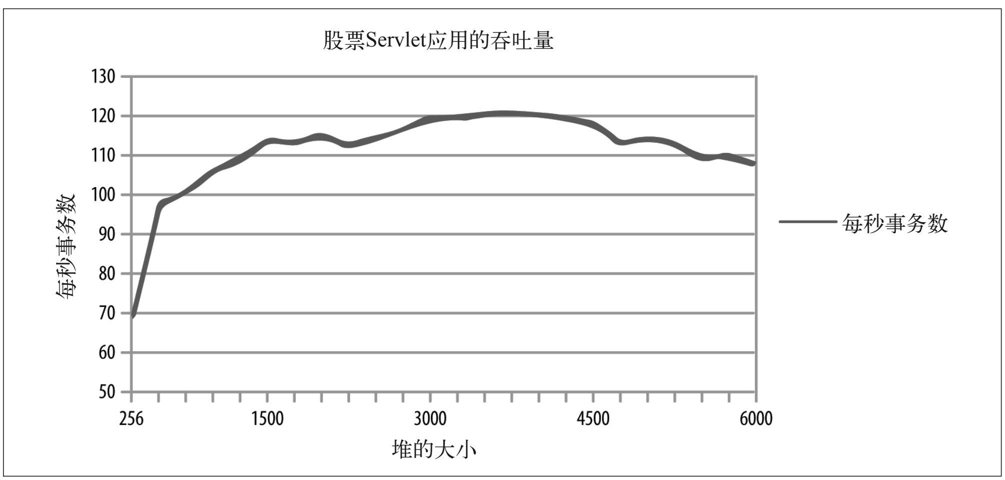


图 6-3：使用不同大小的堆时吞吐量的变化

为了达到停顿时间的指标，Throughput 收集器的自适应调整会重新分配堆（以及代）的大小。使用这些标志可以设置相应的性能指标：`-XX:MaxGCPauseMillis=`*N* 和 `-XX:GCTimeRatio=`*N*。

`MaxGCPauseMillis` 标志用于设定应用可承受的最大停顿时间。我们可以将其设置为 `0` 或者一些非常小的值，譬如 `50` 毫秒。请注意，这个标志设定的值同时影响 Minor GC 和 Full GC。如果设置的值非常小，那么应用的老年代最终就会非常小：譬如，你设定该参数希望应用在 `50` 毫秒内完成垃圾回收，这将会触发非常频繁的 Full GC，对应用程序的性能而言

将是灾难性的。因此，设定该值时，请尽量保持理性，将该值设定为可达到的合理值。缺省情况下，我们不设定该参数。

`GCTimeRatio` 标志可以设置你希望应用程序在垃圾回收上花费多少时间（与应用线程的运行时间相比较）。它是一个百分比，因此 N 值的计算稍微有些复杂。将 N 值代入下面的公式可以计算出理想情况下应用线程的运行时间所占的百分比：

$$ThroughputGoal = 1 - \frac{1}{(1 + GCTimeRatio)}$$

`GCTimeRatio` 的默认值是 99。将该值代入公式能得到 0.99，这意味着应用程序的运行时间占总时间的 99%，只有 1% 的时间消耗在垃圾回收上。但是，不要被列出的默认值搞糊涂。譬如，`GCTimeRatio` 设置为 95 并不意味着会使用总时间的 5% 去做垃圾回收；它表示的是最多会使用总时间的 1.94% 去做垃圾回收。

先确定你期望应用程序线程工作的时间（譬如 95%），再根据下面这个公式计算 GCTimeRatio 是一个更容易操作的方法。

$$GCTimeRatio = \frac{Throughput}{(1 - Throughput)}$$

对于 95%（0.95）的吞吐量目标，利用该公式计算出的 `GCTimeRatio` 是 19。

JVM 使用这两个标志在堆的初始值（`-Xms`）和最大值（`-Xmx`）之间设置堆的大小。`MaxGCPauseMillis` 标志的优先级最高：如果设置了这个值，新生代和老年代会随之进行调整，直到满足对应停顿时间的目标。一旦这个目标达成，堆的总容量就开始逐渐增大，直到运行时间的比率达到设定值。这两个目标都达成后，JVM 会尝试缩减堆的大小，尽可能以最小的堆大小来满足这两个目标。

由于默认情况不设置停顿时间目标，通常自动堆调整的效果是堆（以及代空间）的大小会持续增大，直到满足设置的 `GCTimeRatio` 目标。不过，在实际操作中，该标志的默认设置已经相当优化了。每个人的使用经验各有不同，但是根据我以往的经验，如果应用程序在垃圾回收上消耗总时间的 3% 至 6%，其效果会是相当不错的。有些时候，我甚至会在内存严重受限的环境中调优应用程序的性能；这些应用通常会在垃圾回收上消耗 10% 至 15% 的时间。垃圾回收对这些应用程序的性能影响巨大，不过整体的性能目标还是能够达到的。

因此，依据应用程序的性能目标，最佳的配置也有所不同。本例没有其他的性能目标，我从时间百分比 19（垃圾收集时间占整个时间的 5% 左右）开始。

表 6-1 展示的是一个应用，仅需要一个容量较小的堆，也很少做 GC（这是一个运行在 GlassFish 服务器上的股票 Servlet 程序，它不保持会话状态，几乎没有长期活跃的对象），使用动态调整后的效果。

表6-1：使用动态GC调整的效果

垃圾收集参数	最终堆的大小	垃圾收集时间所占百分比	OPS
默认值	649 MB	0.9%	9.2
`MaxGCPauseMillis=50ms`	560 MB	1.0%	9.2
`Xms=Xmx = 2048m`	2 GB	0.04%	9.2

默认情况下，堆容量的最小值是 64 MB，最大为 2 GB（因为这台机器配备了 8GB 的物理内存）。这时，`GCTimeRatio` 就如我们预期的那样工作得很好：堆的容量动态地调整到了 649 MB，这时应用程序在垃圾回收上只花费了大约 1% 的总运行时间。

在这个例子中，如果设置了 `MaxGCPauseMillis` 参数，JVM 为了达到停顿时间的目标，这之后就开始逐步减小堆的大小。由于本例中垃圾收集器不需要做太多的工作，堆的调整进行得很顺利，调整之后的堆仍能维持只消耗 1% 的时间在垃圾回收上，同时保持跟之前同样 9.2 次每秒（OPS）的吞吐量。

最后你会发现堆并不是越大越好——使用大小为 2 GB 的堆可以减少应用程序在垃圾回收上消耗的时间，但是这个例子中垃圾回收并不是影响性能的决定性因素，因此也无法提高程序的吞吐量。正如之前提到的，在错误的方向上投注精力进行优化无法提升应用的性能。

同样的应用程序，如果改变了行为，需要保持每个用户前 50 个请求的会话状态，垃圾收集器的工作量就会大大增加。表 6-2 展示了这种情况下的取舍。

表6-2：动态调整的效果

垃圾收集参数	最终堆的大小	垃圾收集时间所占百分比	OPS
默认值	1.7 GB	9.3%	8.4
`MaxGCPauseMillis=50ms`	588 MB	15.1%	7.9
`Xms=Xmx=2048m`	2 GB	5.1%	9.0
`Xms=3560m;MaxGCRatio=19`	2.1 GB	8.8%	9.0

如果测试中应用程序消耗了大量的时间在垃圾回收上，情况就不一样了。这时，JVM 将无法达到设定的吞吐量目标，即只花总运行时间的 1% 在垃圾回收上，它会拼命地尝试各种途径来达到设定的目标，最终使用的堆空间为 1.7 GB。

如果设定的停顿时间目标不切实际，情况会更糟。为了让垃圾收集的时间控制在 50 毫秒以内，我们需要将堆的大小保持在 588 MB 以下，但这又意味着垃圾收集的频率变得过于频繁。最终，应用程序的吞吐量会因此显著降低。这种情况下，让 JVM 使用整个堆的容量，即将堆的初始值和最大值都设置成 2 GB 能获得更好的性能。

最终，我们通过努力将堆的大小设置得比较合理，将时间比率目标也设置得比较现实（5%），表的最后一行展示了这时的结果。JVM 通过自身的计算确定了 2 GB 是最优的堆大小，达到了设定的吞吐量目标，这与手工调整的效果几乎一致。

快速小结

1. 采用动态调整是进行堆调优极好的入手点。对很多的应用程序而言，采用动态调整就已经足够，动态调整的配置能够有效地减少 JVM 的内存使用。
2. 静态地设置堆的大小也可能获得最优的性能。设置合理的性能目标，让 JVM 根据设置确定堆的大小是学习这种调优很好的入门课程。

6.2　理解CMS收集器

CMS 收集器有 3 种基本的操作，分别是：

- CMS 收集器会对新生代的对象进行回收（所有的应用线程都会被暂停）；
- CMS 收集器会启动一个并发的线程对老年代空间的垃圾进行回收；
- 如果有必要，CMS 会发起 Full GC。

图 6-4 展示了使用 CMS 回收新生代的情况。

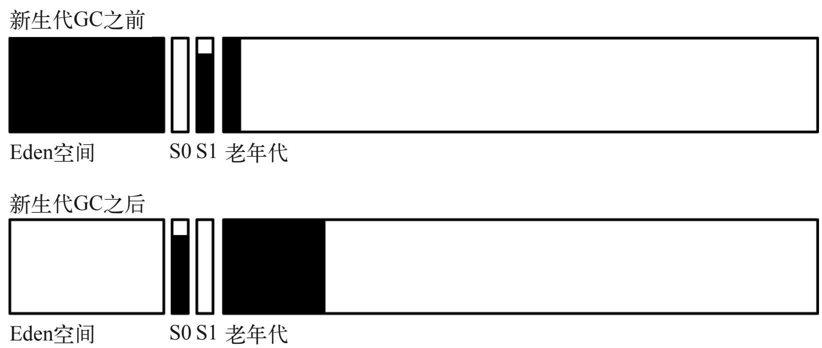


图 6-4：使用 CMS 收集器回收新生代空间

CMS 收集器的新生代垃圾收集与 Throughput 收集器的新生代垃圾收集非常相似：对象从 Eden 空间移动到 Survivor 空间，或者移动到老年代空间。CMS 收集的 GC 日志也非常相似：

```
89.853: [GC 89.853: [ParNew: 629120K->69888K(629120K), 0.1218970 secs]
            1303940K->772142K(2027264K), 0.1220090 secs]
            [Times: user=0.42 sys=0.02, real=0.12 secs]
```

这时的新生代空间大小为 629 MB；垃圾回收之后变成了 69 MB（位于 Survivor 空间）。与 Throughput 收集器的日志类似，整个堆的大小为 2027 MB，其中 772 MB 在垃圾回收之后依然被占用。虽然并行的 GC 线程使用了 0.42 秒的 CPU 时间，但整个垃圾回收过程仅耗时 0.12 秒。并发的垃圾回收周期如图 6-5 所示。

JVM 会依据堆的使用情况启动并发回收。当堆的占用达到某个程度时，JVM 会启动后台线程扫描堆，回收不用的对象。扫描结束的时候，堆的状况就像这幅图中最后一列所描述的情况一样。请注意，如果使用 CMS 回收器，老年代空间不会进行压缩整理：老年代空间由已经分配对象的空间和空闲空间共同组成。新生代垃圾收集将对象由 Eden 空间挪到老年代空间时，JVM 会尝试使用那些空闲的空间来保存这些晋升的对象。

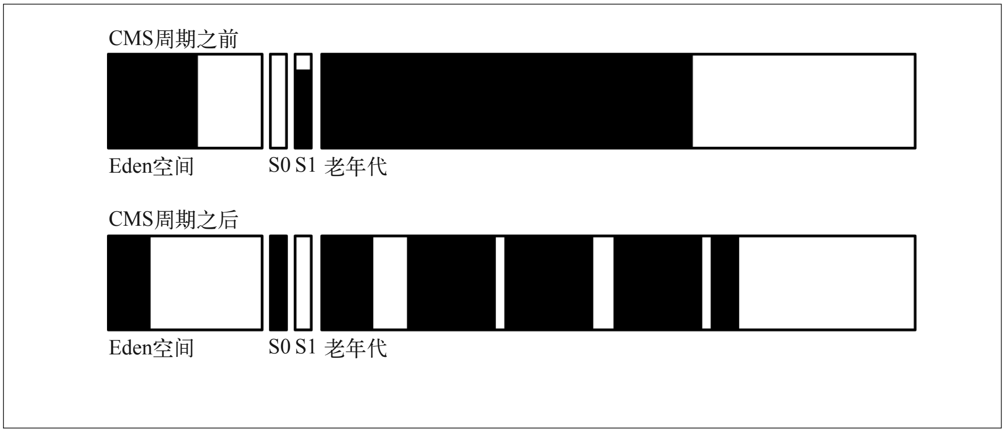


图 6-5：由 CMS 收集器完成的并发垃圾收集

通过 GC 日志，我们看到回收过程划分成了好几个阶段。虽然主要的并发回收（Concurrent Cycle）阶段都使用后台线程进行工作，有些阶段还是会暂停所有的应用线程，并因此引入短暂的停顿。

并发回收由“初始标记”阶段开始，这个阶段会暂停所有的应用程序线程：

```
89.976: [GC [1 CMS-initial-mark: 702254K(1398144K)]
                772530K(2027264K), 0.0830120 secs]
                [Times: user=0.08 sys=0.00, real=0.08 secs]
```

这个阶段的主要任务是找到堆中所有的垃圾回收根节点对象。从第一组数据中可以看到这个例子中对象占用了老年代空间 1398 MB 中的 702 MB 空间。第二组数据显示整个堆的大小为 2027 MB，其中 772 MB 被占用。应用程序线程在这个 CMS 回收周期中被暂停了 0.08 秒。

下一个阶段是“标记阶段”，这个阶段中应用程序线程可以持续运行，不会被中断。GC 日志中，这个阶段的标识如下：

```
90.059: [CMS-concurrent-mark-start]
90.887: [CMS-concurrent-mark: 0.823/0.828 secs]
                [Times: user=1.11 sys=0.00, real=0.83 secs]
```

标识阶段耗时 0.83 秒（以及 1.11 秒的 CPU 时间）。由于这个阶段进行的工作仅仅是标记，不会对堆的使用情况产生实质性的改变，所以没有任何相关的数据输出。如果这个阶段还有数据输出，很可能是由于这 0.83 秒内新生代对象的分配导致了堆的增长，因为应用程序线程还在持续运行着。

然后是“预清理”阶段，这个阶段也是与应用程序线程的运行并发进行的：

```
90.887: [CMS-concurrent-preclean-start]
90.892: [CMS-concurrent-preclean: 0.005/0.005 secs]
                [Times: user=0.01 sys=0.00, real=0.01 secs]
```

接下来的是“重新标记”阶段，这个阶段涵盖了多个操作：

```
90.892: [CMS-concurrent-abortable-preclean-start]
92.392: [GC 92.393: [ParNew: 629120K->69888K(629120K), 0.1289040 secs]
                1331374K->803967K(2027264K), 0.1290200 secs]
                [Times: user=0.44 sys=0.01, real=0.12 secs]
94.473: [CMS-concurrent-abortable-preclean: 3.451/3.581 secs]
                [Times: user=5.03 sys=0.03, real=3.58 secs]

94.474: [GC[YG occupancy: 466937 K (629120 K)]
        94.474: [Rescan (parallel) , 0.1850000 secs]
        94.659: [weak refs processing, 0.0000370 secs]
        94.659: [scrub string table, 0.0011530 secs]
                [1 CMS-remark: 734079K(1398144K)]
                1201017K(2027264K), 0.1863430 secs]
        [Times: user=0.60 sys=0.01, real=0.18 secs]
```

且慢，CMS 收集不是只执行一次预清理阶段吗？这个“可中断预清理”（abortable preclean）阶段是做什么的呢？

使用可中断预清理阶段是由于标记阶段（严格说起来，它应该是最后的输出项）不是并发的，所有的应用线程进入标记阶段后都会被暂停。如果新生代收集刚刚结束，紧接着就是一个标记阶段的话，应用线程会遭遇 2 次连续的停顿操作，CMS 收集器希望避免这样的情况发生。使用可中断预清理阶段的目的就是希望尽量缩短停顿的长度，避免连续的停顿。

因此，可中断预清理阶段会等到新生代空间占用到 50% 左右时才开始。理论上，这时离下一次新生代收集还有半程的距离，给了 CMS 收集器最好的机会避免发生连续停顿。这个例子中，可中断预清理阶段在 90.8 秒开始，等待常规的新生代收集开始花了 1.5 秒（根据日志的记录，92.392 秒开始）。CMS 收集器根据以往的历史记录推算下一次新生代垃圾收集可能持续的时间。[1] 这个例子中，CMS 收集器计算出的时长大约是 4.2 秒。所以 2.1 秒之后（即 94.4 秒），CMS 收集器停止了预清理阶段（这种行为被称为“放弃”了这次回收，不过这可能是唯一能停止该次回收的方式）。这之后，CMS 回收器终于开始了标记阶段的工作执行，标记阶段的回收工作将应用程序线程暂停了 0.18 秒（在可中断预清理过程中，应用程序线程不会被暂停）。

接下来是另一个并发阶段——清除（sweep）阶段：

```
94.661: [CMS-concurrent-sweep-start]
95.223: [GC 95.223: [ParNew: 629120K->69888K(629120K), 0.1322530 secs]
                999428K->472094K(2027264K), 0.1323690 secs]
                [Times: user=0.43 sys=0.00, real=0.13 secs]
95.474: [CMS-concurrent-sweep: 0.680/0.813 secs]
                [Times: user=1.45 sys=0.00, real=0.82 secs]
```

这个阶段耗时 0.82 秒，回收线程与应用程序线程并发运行。碰巧这次的并发 - 清除过程被新生代垃圾回收中断了。新生代垃圾回收与清除阶段并没有直接的联系，将这个例子保留在这里是为了说明新生代的垃圾收集与老年代的垃圾收集可以并发进行。从图 6-5 中可以看到，新生代的状态在并发收集的过程中发生了变化——清除过程中新生代可能发生了多次垃圾收集（至少发生了一次新生代垃圾收集，因为可中断的预清理至少会经历一次新生

注 1：此处原文可能有误。根据新生代垃圾收集器的历史能得到的应该是新生代垃圾收集的持续时间，而非准确的什么时候发生。——译者注

代垃圾收集）。

接下来是并发重置（concurrent reset）阶段：

```
95.474: [CMS-concurrent-reset-start]
95.479: [CMS-concurrent-reset: 0.005/0.005 secs]
        [Times: user=0.00 sys=0.00, real=0.00 secs]
```

这是并发运行的最后一个阶段；CMS 垃圾回收的周期至此告终，老年代空间中没有被引用的对象被回收（此时堆的状况如图 6-5 所示）。遗憾的是，我们无法从日志中了解到底有多少对象被回收；重置阶段的日志也没有提供更多的信息，最后还有多少堆空间被占用不得而知。为了发掘这些信息，我们尝试从新生代垃圾收集日志中找到一些蛛丝马迹，如下所示：

```
98.049: [GC 98.049: [ParNew: 629120K->69888K(629120K), 0.1487040 secs]
                1031326K->504955K(2027264K), 0.1488730 secs]
```

与 89.853 秒时（即 CMS 回收周期开始之前）老年代空间的占用情况相比较，那时的空间占用大约是 703 MB（整个堆的占用为 772 MB，其中包含 69 MB 的 Survivor 空间占用，因此老年代占用了剩下的 703 MB）。到 98.049 秒，垃圾收集结束，老年代空间占用大约为 504 MB，由此可以计算出 CMS 周期回收了大约 199 MB 的内存。

如果一切顺利，这些就是 CMS 垃圾回收会经历的周期，以及所有可能出现在 CMS 垃圾收集日志中的消息。不过，事实并不是这么简单，我们还需要查看另外三种消息，出现这些日志表明 CMS 垃圾收集碰到了麻烦。首当其冲的是并发模式失效（concurrent mode failure）：

```
267.006: [GC 267.006: [ParNew: 629120K->629120K(629120K), 0.0000200 secs]
        267.006: [CMS267.350: [CMS-concurrent-mark: 2.683/2.804 secs]
        [Times: user=4.81 sys=0.02, real=2.80 secs]
        (concurrent mode failure):
        1378132K->1366755K(1398144K), 5.6213320 secs]
        2007252K->1366755K(2027264K),
        [CMS Perm : 57231K->57222K(95548K)], 5.6215150 secs]
        [Times: user=5.63 sys=0.00, real=5.62 secs]
```

新生代发生垃圾回收，同时老年代又没有足够的空间容纳晋升的对象时，CMS 垃圾回收就会退化成 Full GC。所有的应用线程都会被暂停，老年代中所有的无效对象都被回收，释放空间之后老年代的占用为 1366 MB——这次操作导致应用程序线程停顿长达 5.6 秒。这个操作是单线程的，这就是为什么它耗时如此之长的原因之一（这也是为什么发生并发模式失效比堆的增长更加恶劣的原因之一）。

第二个问题是老年代有足够的空间可以容纳晋升的对象，但是由于空闲空间的碎片化，导致晋升失败：

```
6043.903: [GC 6043.903:
        [ParNew (promotion failed): 614254K->629120K(629120K), 0.1619839 secs]
        6044.217: [CMS: 1342523K->1336533K(2027264K), 30.7884210 secs]
        2004251K->1336533K(1398144K),
        [CMS Perm : 57231K->57231K(95548K)], 28.1361340 secs]
        [Times: user=28.13 sys=0.38, real=28.13 secs]
```

在这个例子中，CMS 启动了新生代垃圾收集，判断老年代似乎有足够的空闲空间可以容纳所有的晋升对象（否则，CMS 收集器会报告发生并发模式失效）。这个假设最终被证明是错误的：由于老年代空间的碎片化（或者，不太贴切地说，由于晋升实际要占用的内存超过了 CMS 收集器的判断），CMS 收集器无法晋升这些对象。

因此，CMS 收集器在新生代垃圾收集过程中（所有的应用线程都被暂停时），对整个老年代空间进行了整理和压缩。好消息是，随着堆的压缩，碎片化问题解决了（至少在短期内不是问题了）。不过随之而来的是长达 28 秒的冗长的停顿时间。由于需要对整个堆进行整理，这个时间甚至比 CMS 收集器遭遇并发模式失效的时间还长的多，因为发生并发模式失效时，CMS 收集器只需要回收堆内无用的对象。这时的堆就像刚由 Throughput 收集器做完 Full GC 一样（如图 6-2）：新生代空间完全空闲，老年代空间也已经整理过。

最终，CMS 收集的日志中可能只有一条 Full GC 的记录，不含任何常规并发垃圾回收的日志。

```
279.803: [Full GC 279.803:
                [CMS: 88569K->68870K(1398144K), 0.6714090 secs]
                558070K->68870K(2027264K),
                [CMS Perm : 81919K->77654K(81920K)],
                0.6716570 secs]
```

永久代空间用尽，需要回收时，就会发生这样的状况；应注意到，CMS 收集后永久代空间大小减小了。Java 8 中，如果元空间需要调整，也会发生同样的情况。默认情况下，CMS 收集器不会对永久代（或元空间）进行收集，因此，它一旦被用尽，就需要进行 Full GC，所有没有被引用的类都会被回收。CMS 高级调优一节会有针对性地介绍如何解决这种问题。

快速小结

1. CMS 垃圾回收有多个操作，但是期望的操作是 Minor GC 和并发回收（concurrent cycle）。
2. CMS 收集过程中的并发模式失效以及晋升失败的代价都非常昂贵；我们应该尽量调优 CMS 收集器以避免发生这些情况。
3. 默认情况下 CMS 收集器不会对永久代进行垃圾回收。

6.2.1　针对并发模式失效的调优

调优 CMS 收集器时最要紧的工作就是要避免发生并发模式失效以及晋升失败。正如我们在 CMS 垃圾收集日志中看到的那样，发生并发模式失效往往是由于 CMS 不能以足够快的速度清理老年代空间：新生代需要进行垃圾回收时，CMS 收集器计算发现老年代没有足够的空闲空间可以容纳这些晋升对象，不得不先对老年代进行垃圾回收。

初始时老年代空间中对象是一个接一个整齐有序排列的。当老年代空间的占用达到某个程度（默认值为 70%）时，并发回收就开始了。一个 CMS 后台线程开始扫描老年代空间，寻找无用的垃圾对象时，竞争就开始了：CMS 收集器必须在老年代剩余的空间（30%）用尽之前，完成老年代空间的扫描及回收工作。如果并发回收在这场速度的比赛中失利，CMS 收集器就会发生并发模式失效。

有以下途径可以避免发生这种失效。

- 想办法增大老年代空间，要么只移动部分的新生代对象到老年代，要么增加更多的堆空间。
- 以更高的频率运行后台回收线程。
- 使用更多的后台回收线程。

> **自适应调优和 CMS 垃圾搜集**
>
> CMS 收集器使用两个配置 `MaxGCPauseMllis=N` 和 `GCTimeRatio=N` 来确定使用多大的堆和多大的代空间。
>
> CMS 收集与其他的垃圾收集方法一个显著的不同是除非发生 Full GC，否则 CMS 的新生代大小不会作调整。由于 CMS 的目标是尽量避免 Full GC，这意味着使用精细调优的 CMS 收集器的应用程序永远不会调整它的新生代大小。
>
> 程序启动时可能频发并发模式失效，因为 CMS 收集器需要调整堆和永久代（或者元空间）的大小。使用 CMS 收集器，初始时采用一个比较大的堆（以及更大的永久代 / 元空间）是一个很好的主意，这是一个特例，增大堆的大小反而帮助避免了那些失效。

如果有更多的内存可用，更好的方案是增加堆的大小，否则可以尝试调整后台线程运行的方式来解决这个问题。

1. 给后台线程更多的运行机会

为了让 CMS 收集器赢得这场比赛，方法之一是更早地启动并发收集周期。显然地，CMS 收集器在老年代空间占用达到 60% 时启动并发周期，这和老年代空间占用到 70% 时才启动相比，前者完成垃圾收集的几率更大。为了实现这种配置，最简单的方法是同时设置下面这两个标志：`-XX:CMSInitiatingOccupancyFraction=N` 和 `-XX:+UseCMSInitiatingOccupancyOnly`。同时使用这两个参数能帮助 CMS 更容易地进行决策：如果同时设置这两个标志，那么 CMS 就只依据设置的老年代空间占用率来决定何时启动后台线程。默认情况下，`UseCMSInitiatingOccupancyOnly` 标志的值为假，CMS 会使用更复杂的算法判断什么时候启动并行收集线程。如果有必要提前启动后台线程，推荐使用最简单的方法，即将 `UseCMSInitiatingOccupancyOnly` 标志的值设置为真。

`CMSInitiatingOccupancyFraction` 参数值的调整可能需要多次迭代才能确定。如果开启了 `UseCMSInitiatingOccupancyOnly` 标志，`CMSInitiatingOccupancyFraction` 的默认值就被置为 70，即 CMS 会在老年代空间占用达到 70% 时启动并发收集周期。

对特定的应用程序，该标志的更优值可以根据 GC 日志中 CMS 周期首次启动失败时的值得到。具体方法是，在垃圾回收日志中寻找并发模式失效，找到后再反向查找 CMS 周期最近的启动记录。日志中含有 CMS-initial-mark 信息的一行包含了 CMS 周期启动时，老年代空间的占用情况如下所示：

```
89.976: [GC [1 CMS-initial-mark: 702254K(1398144K)]
                772530K(2027264K), 0.0830120 secs]
                [Times: user=0.08 sys=0.00, real=0.08 secs]
```

在这个例子中，根据日志的输出，我们可以判断该时刻老年代空间的占用率为 50%（老年代空间大小为 1398 MB，其中 702 MB 被占用）。不过这个值还不够早，因此我们需要调整 `CMSInitiatingOccupancyFraction` 将其值设定为小于 50 的某个值。（虽然 `CMSInitiatingOccupancyFraction` 的默认值为 70，不过这个例子中没有开启 `UseCMSInitiatingOccupancyOnly` 标志，所以例子中 CMS 收集器在老年代空间占用达到 50% 时启动了 CMS 后台线程。）

了解了 `CMSInitiatingOccupancyFraction` 的工作原理之后，你可能会有疑问，我们能不能将参数值设置为 0 或者其他比较小的值，让 CMS 的后台线程持续运行。通常我们不推荐进行这样的设置，但是，如果你对其中的取舍非常了解，适当地妥协也是可以接受的。

这其中的第一个取舍源于 CPU：CMS 后台线程会持续运行，它们会消耗大量的 CPU 时钟——每个 CMS 后台线程运行时都会 100% 地占用一颗 CPU。多个 CMS 线程同时运行时还会有短暂的爆发，机器的总 CPU 使用因此也会暴涨。如果这些线程都是毫无目的地持续运行，只会白白浪费宝贵的 CPU 资源。

另一方面，这并不是说使用了过多的 CPU 周期就是问题。后台的 CMS 线程需要时必须运行，即使在最好的情况下，这也是很难避免的。因此，机器必须预留足够的 CPU 周期来运行这些 CMS 线程。所以规划机器时，你必须考虑留出余量给这部分 CPU 的使用。

CMS 周期中，如果 CMS 后台线程没有运行，这些 CPU 时钟可以用于运行其他的应用吗？通常不会。如果还有另一个应用也在使用同一个时钟周期，它没有途径了解何时 CMS 线程会运行。因此，应用程序线程和 CMS 线程会竞争 CPU 资源，而这很可能会导致 CMS 线程的“失速”（lose its race）。有些时候，通过复杂的操作系统调优，有可能让应用线程以低于 CMS 线程优先级的方式让两种线程在同一个时钟周期内运行，但是这些方法都相当复杂，很容易出错。因此，答案是肯定的，CMS 周期运行得越频繁，CPU 周期越长，如果不这样，这些 CPU 周期就是空闲状态（idle）。

第二个取舍更加重要，它与应用程序的停顿相关。正如我们在 GC 日志中观察到的，CMS 在特定的阶段会暂停所有的应用线程。使用 CMS 收集器的主要目的就是要限制 GC 停顿的影响，因此频繁地运行更多无效的 CMS 周期只能适得其反。CMS 停顿的时间与新生代的停顿时间比起来要短得多，应用线程甚至可能感受不到这些额外的停顿——这也是一种取舍，我们是要避免额外的停顿还是要减少发生并发模式失败的几率。不过，正如我们前面提到的，持续地运行后台 GC 线程所造成的停顿可能会导致总体的停顿，而这最终会降低应用程序的性能。

除非这些取舍都能接受，否则不要将 `CMSInitiatingOccupancyFraction` 参数的值设置得比堆内的活跃数据数还多，至少要少 10% 到 20%。[2]

2. 调整CMS后台线程

每个 CMS 后台线程都会 100% 地占用机器上的一颗 CPU。如果应用程序发生并发模式失效，同时又有额外的 CPU 周期可用，可以设置 `-XX:ConcGCThreads=N` 标志，增加后台线程的数

注 2：原文可能有误，`CMSInitiatingOccupancyFraction` 应该设置得比活跃数据小才能提前触发 CMS 周期。——译者注

目。默认情况下，ConcGCThreads 的值是依据 ParallelGCThreads 标志的值计算得到的：

```
ConcGCThreads = (3 + ParallelGCThreads) / 4
```

上述计算使用整数计算方法，这意味着如果 ParallelGCThreads 的取值区间在 1 到 4，ConcGCThread 的值就为 1，如果 ParallelGCThreads 的取值在 5 到 8 之间，ConcGCThreads 的值就为 2，以此类推。

调整这一标志的要点在于判断是否有可用的 CPU 周期。如果 ConcGCThreads 标志值设置的偏大，垃圾收集会占用本来能用于运行应用线程的 CPU 周期；最终效果上，这种配置会导致应用程序些微的停顿，因为应用程序线程需要等待再次在 CPU 上继续运行的机会。

除此之外，在一个配备了大量 CPU 的系统上，ConcGCThreads 参数的默认值可能偏大。如果没有频繁遭遇并发模式失败，可以考虑减少后台线程数，释放这部分 CPU 周期用于应用线程的运行。

快速小结

1. 避免发生并发模式失效是提升 CMS 收集器处理能力、获得高性能的关键。
2. 避免并发模式失效（如果有可能的话）最简单的方法是增大堆的容量。
3. 否则，我们能进行的下一个步骤就是通过调整 CMSInitiatingOccupancyFraction 参数，尽早启动并发后台线程的运行。
4. 另外，调整后台线程的数目对解决这个问题也有帮助。

6.2.2 CMS收集器的永久代调优

从例子的 CMS 垃圾收集日志中我们发现，如果永久代需要进行垃圾收集，就会发生 Full GC（如果元空间的大小需要调整也会发生同样的情况）。这往往发生在程序员频繁部署（或者重新部署）应用的服务器上，或者发生在需要频繁定义（或者回收）类的应用中。

默认情况下，Java 7 中的 CMS 垃圾收集线程不会处理永久代中的垃圾，如果永久代空间用尽，CMS 会发起一次 Full GC 来回收其中的垃圾对象。除此之外，还可以开启 -XX:+CMSPermGenSweepingEnabled 标志（默认情况下，该标志的值为 false），开启后，永久代中的垃圾使用与老年代同样的方式进行垃圾收集：通过一组后台线程并发地回收永久代中的垃圾对象。注意，触发永久代垃圾回收的指标与老年代的指标是相互独立的。使用 -XX:CMSInitiatingPermOccupancyFraction=*N* 参数可以指定 CMS 收集器在永久代空间占用比达到设定值时启动永久代垃圾回收线程，这个参数的默认值为 80%。

不过，开启永久代垃圾收集只是整个流程中的一步，为了真正释放不再被引用的类，我们还需要设置 -XX:+CMSClassUnloadingEnabled 标志。否则，即使启用了永久代垃圾回收也只能释放少量的无效对象，类的元数据并不会被释放。由于永久代中大量的数据都是类的元数据，因此启动 CMS 永久代垃圾收集时，这个标志同时也应该开启。

Java 8 中，CMS 收集器默认就会收集元空间中不再载入的类。如果由于某些原因，你希望关闭这一功能，可以通过 -XX:-CMSClassUnloadingEnabled 标志进行关闭（默认情况下这个标志是开启的，即该值为 true）。

6.2.3 增量式CMS垃圾收集

这一章中我们多次提到了这样一个事实：为了进行有效的 CMS 垃圾收集，需要消耗额外的 CPU 处理资源。如果你只有一个单 CPU 的机器，或者你有多个非常忙碌的 CPU，但是希望使用低延迟的垃圾收集器，这时有什么好的建议呢？

> **增量式 CMS 垃圾收集在 Java 8 中已经不推荐使用**
>
> 增量式 CMS 垃圾收集（iCMS）在 Java 8 中已经不推荐使用了，不过暂时还保留在其中，但是在 Java 9 中很可能会被移除。
>
> 使用增量式 CMS 垃圾收集的主要好处是后台线程会间歇性地停顿，让出一部分 CPU 给应用程序线程运行，从而使得 CMS 收集器即使在只配备了有限 CPU 资源的机器上也能运行。随着多核技术的发展，多处理器几乎已经成为所有系统的标准配置（连我的手机都装载了 4 核的 CPU 芯片），这使得 iCMS 存在的意义变得不再那么重要。
>
> 如果系统确实只配备了极其有限的 CPU，作为替代方案，可以考虑使用 G1 收集器——因为 G1 收集器的后台线程在垃圾收集的过程中也会周期性地暂停，客观上减少了与应用线程竞争 CPU 资源的情况。

这些情况下，使用 CMS 收集器进行增量式的垃圾收集，即只要有后台线程运行（同一个时刻处于运行状态的线程数不应该超过一个），垃圾收集器就不会马上对整个堆进行垃圾收集。这个后台线程间断性地暂停，有助于整个系统吞吐量的提高，因为更多的 CPU 处理资源让给了应用线程的运行。当然，如果 CMS 收集线程一旦运行起来，还是会与应用程序线程争夺有限的 CPU 处理周期。

指定 `-XX:+CMSIncrementalMode` 标志可以开启增量式 CMS 垃圾收集。通过改变标志 `-XX:CMSIncrementalSafetyFactor=`*`N`*、`-XX:CMSIncrementalDutyCycleMin=`*`N`* 和 `-XX:CMSIncrementalPacing` 可以控制垃圾收集后台线程为应用程序线程让出多少 CPU 周期。

增量式 CMS 垃圾收集依据责任周期（duty cycle）原则进行工作，这个原则决定了 CMS 垃圾收集器的后台线程在释放 CPU 周期给应用线程之前，每隔多长时间扫描一次堆。从操作系统的层次上看，CMS 垃圾收集器的后台线程已经和应用的线程发生了竞争（通常是基于时间片的）。换个角度看，这些标志实际控制着主动暂停运行、释放资源给应用线程运行之前，后台线程持续运行的时间。

责任周期的时间长度是以新生代相邻两次垃圾收集之间的时间长度计算得出的；默认情况下，增量式 CMS 垃圾收集持续的时间是该时长的 20% 左右（至少初始时是这个值，不过 CMS 会不断调整该值以适应不断晋升到老年代的对象数目）。如果这个时间不够长，就会发生并发模式失效（以及 Full GC）。我们的目标就是通过调整增量式 CMS 垃圾收集，避免发生这种 GC（或者尽量减少它们发生的频率）。

我们从调整增大 `CMSIncrementalSafetyFactor` 参数入手，这个参数设置是增加到默认责任周期的时间百分比。责任周期的默认值是 10%，默认情况下，安全因子（`safety factor`）的值是再增加 10%（这样默认的初始责任周期所占用的时间百分比就变成了 20%）。通过

增大安全因子（最大可以增加到 90，不过这会导致增量周期占用所有的时间），可以让后台线程有更多的运行时间。

除此之外，如果参数 `CMSIncrementalDutyCycleMin` 设置得比默认值（10）更大也可以调整责任周期的长度。不过这个参数值会受 JVM 自动调节机制的影响，因为 JVM 的自动调节机制会监控由新生代晋升到老年代的对象数并进行相应的调节。所以，即使增大这个值，JVM 可能还是会依据自身的判断，即增量式垃圾收集运行不需要运行得过于频繁，而减小这个参数的值。如果应用程序运行时操作有爆发式的波峰，通过自动调节机制计算出的结果通常不准确，你需要显式地设置责任周期，同时调整 `CMSIncrementalDutyCycle` 标志关闭自动参数调节（`CMSIncrementalDutyCycle` 的值默认为真，即开启）。

快速小结

1. 应用在 CPU 资源受限的机器上运行，同时又要求较小的停顿，这时使用增量式 CMS 收集器是一个不错的选择。
2. 通过责任周期可以调整增量式 CMS 收集器；增加责任周期的运行时间可以避免 CMS 收集器发生并发模式失效。

6.3 理解G1垃圾收集器

G1 垃圾收集器是一种工作在堆内不同分区上的并发收集器。分区（region）既可以归属于老年代，也可以归属于新生代（默认情况下，一个堆被划分成 2048 个分区），同一个代的分区不需要保持连续。为老年代设计分区的初衷是我们发现并发后台线程在回收老年代中没有引用的对象时，有的分区垃圾对象的数量很多，另一些分区的垃圾对象相对较少。虽然分区的垃圾收集工作实际仍然会暂停应用程序线程，不过由于 G1 收集器专注于垃圾最多的分区，最终的效果是花费较少的时间就能回收这些分区的垃圾。这种只专注于垃圾最多分区的方式就是 G1 垃圾收集器名称的由来，即首先收集垃圾最多的分区。

不过这一算法并不适用于新生代的分区：新生代进行垃圾回收时，整个新生代空间要么被回收，要么被晋升（对象被移动到 Survivor 空间，或者移动到老年代）。新生代也采用分区机制的部分原因，是因为采用预定义的分区能够便于代的大小调整。

G1 收集器的收集活动主要包括 4 种操作：

- 新生代垃圾收集；
- 后台收集，并发周期；
- 混合式垃圾收集；
- 以及必要时的 Full GC。

我们会依次讨论每一种操作，首先讨论的是 G1 收集器的新生代垃圾收集，如图 6-6 所示。

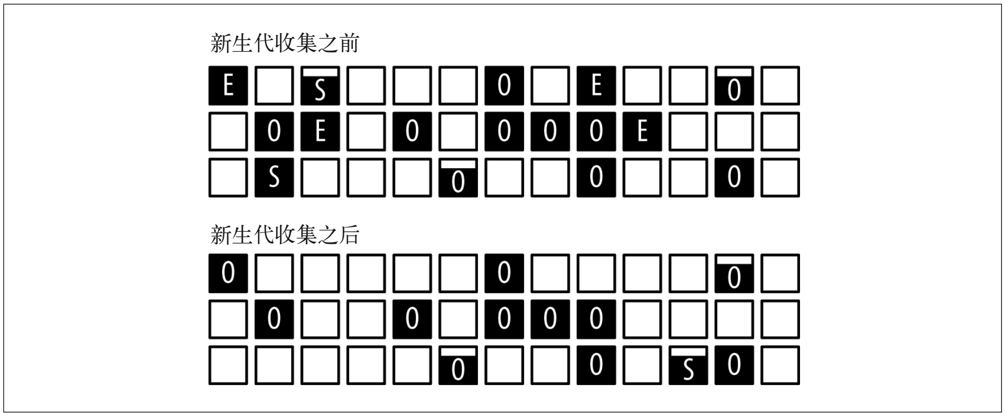


图 6-6：用 G1 垃圾收集器的新生代收集前后对比

图中的每一个小方块都代表一个 G1 的分区。分区中黑色的区域代表数据，每个分区中的字母表示该区域属于哪个代（[E] 代表 Eden 空间，[O] 代表老年代，[S] 代表 Survivor 空间）。空的分区不属于任何一个代；需要的时候 G1 收集器会强制指定这些空的分区用于任何需要的代。

Eden 空间耗尽会触发 G1 垃圾收集器进行新生代垃圾收集（这个例子中，标识为 Eden 的 4 个分区填满之后就会触发新生代收集）。新生代收集之后不会有新的分区马上分配到 Eden 空间，因为这时 Eden 空间为空。不过至少会有一个分区分配到 Survivor 空间（这个例子中，Survivor 空间被部分填满），一部分数据会移动到老年代。

G1 垃圾收集器中，新生代垃圾收集的日志与其他的收集器略有不同。与往常一样，我们可以使用 `PrintGCDetails` 输出例子的垃圾回收日志，不过 G1 收集的日志要详细得多。这里仅仅列出了例子中重要的几行。

下面是新生代垃圾收集的标准流程：

```
23.430: [GC pause (young), 0.23094400 secs]
...
   [Eden: 1286M(1286M)->0B(1212M)
        Survivors: 78M->152M Heap: 1454M(4096M)->242M(4096M)]
   [Times: user=0.85 sys=0.05, real=0.23 secs]
```

这里新生代垃圾收集的 Real 时间消耗是 0.23 秒，这期间，垃圾收集线程消耗了 0.85 秒的 CPU 时间，1286 MB 的对象移出了 Eden 空间（Eden 空间的大小调整到了 1212 MB）；这其中的 74 MB 移动到了 Survivor 空间（Survivor 空间的大小从 78 MB 增加到了 152 MB），其余的空间都被垃圾收集器回收掉了。通过观察堆的总占用降低了 1212 MB 我们知道，这些空间被释放了。通常情况下，一部分对象已经从 Survivor 空间移动到老年代空间，如果 Survivor 空间被填满，无法容纳新生代的晋升对象，部分 Eden 空间的对象会被直接晋升到老年代空间——这种情况下，老年代空间的占用也会增加。

图 6-7 是并发 G1 垃圾收集周期（concurrent G1 cycle）开始和结束时的情况。

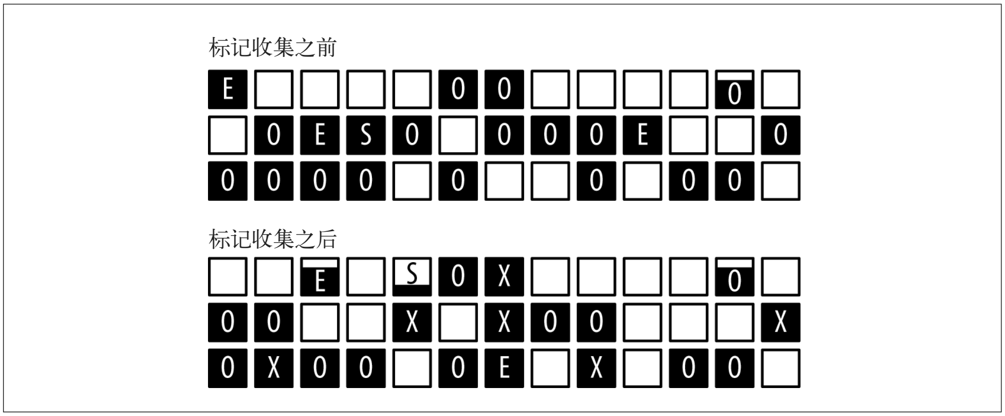


图 6-7：G1 收集器进行的并发垃圾收集

这幅图中有三方面值得我们关注。首先，新生代的空间占用情况发生了变化：在并发周期中，至少有一次（很可能是多次）新生代垃圾收集。因此，在将 Eden 空间中的分区标记为完全释放之前，新的 Eden 分区已经开始分配了。

其次，我们注意到一些分区现在被标记为 X。这些分区属于老年代（注意，它们依然还保持着数据），它们就是标记周期（marking cycle）找出的包含最多垃圾的分区。

最后，我们还要留意老年代（包括标记为 O 或者 X 的分区）的空间占用，在周期结束时实际可能更多。这是因为在标记周期中，新生代的垃圾收集会晋升对象到老年代。除此之外，标记周期中实际不会释放老年代中的任何对象：它仅仅锁定了那些垃圾最多的分区。这些分区中的垃圾数据会在之后的周期中被回收释放。

G1 收集器的并发周期包括多个阶段，其中的一些会暂停所有应用线程，另一些则不会。并发周期的第一个阶段是初始—标记（initial-mark）阶段。这个阶段会暂停所有应用线程——部分源于初始—标记阶段也会进行新生代垃圾收集。

```
50.541: [GC pause (young) (initial-mark), 0.27767100 secs]
    [Eden: 1220M(1220M)->0B(1220M)
        Survivors: 144M->144M Heap: 3242M(4096M)->2093M(4096M)]
    [Times: user=1.02 sys=0.04, real=0.28 secs]
```

同常规的新生代垃圾收集一样，初始—标记阶段中，应用线程被暂停（大约时长 0.28 秒），之后新生代被清空（71 MB 的数据从新生代移到了老年代）。初始—标记阶段的输出日志表明后台并发周期启动。由于初始—标记阶段也需要暂停所有的应用线程，G1 收集器重用了新生代 GC 周期来完成这部分的工作。在新生代垃圾收集中添加初始标记阶段的影响并不大：与之前的垃圾收集相比较，CPU 周期的开销增加了大约 20%，即便如此，停顿时间只有些微的增长（幸运的是，这台机器上有空闲的 CPU 周期可以运行并发 G1 收集线程，否则停顿时间会更长一些）。

接下来，G1 收集器会扫描根分区（root region）：

```
50.819: [GC concurrent-root-region-scan-start]
51.408: [GC concurrent-root-region-scan-end, 0.5890230]
```

这个过程耗时 0.58 秒，不过扫描过程中不需要暂停应用现场，G1 收集器使用后台线程进行扫描工作。不过，这个阶段中不能发生新生代垃圾收集，因此预留足够的 CPU 周期给后台线程运行是非常重要的。如果扫描根分区时，新生代空间刚巧用尽，新生代垃圾收集（会暂停所有的应用线程）必须等待根扫描结束才能完成。效果上，这意味着新生代垃圾收集的停顿时间会更长（远超过正常的耗时）。这种情况在 GC 日志中如下所示：

```
350.994: [GC pause (young)
        351.093: [GC concurrent-root-region-scan-end, 0.6100090]
        351.093: [GC concurrent-mark-start],
        0.37559600 secs]
```

此处 GC 的停顿发生在根分区扫描之前，这意味着 GC 停顿还会继续等待，我们会看到 GC 日志中的相互交织的输出。GC 日志的时间戳显示应用线程等待了大概 100 毫秒——这就是新生代 GC 停顿时间比日志中其他停顿的平均持续时间还长 100 毫秒的原因。这是一个信号，说明你的 G1 收集器需要进行调优，下一节我们将详细讨论这部分内容。

根分区扫描完成后，G1 收集器就进入到并发标记阶段。这个阶段完全在后台运行，阶段启动和停止时在 GC 日志中各会打印一条日志。

```
111.382: [GC concurrent-mark-start]
....
120.905: [GC concurrent-mark-end, 9.5225160 sec]
```

并发标记阶段是可以中断的，所以这个阶段中可能发生新生代垃圾收集。紧接在标记阶段之后的是重新标记（remarking）阶段和正常的清理阶段。

```
120.910: [GC remark 120.959:
        [GC ref-PRC, 0.0000890 secs], 0.0718990 secs]
        [Times: user=0.23 sys=0.01, real=0.08 secs]
120.985: [GC cleanup 3510M->3434M(4096M), 0.0111040 secs]
        [Times: user=0.04 sys=0.00, real=0.01 secs]
```

这几个阶段都会暂停应用线程，虽然暂停的时间通常很短。紧接着是一个额外的并发清理阶段：

```
120.996: [GC concurrent-cleanup-start]
120.996: [GC concurrent-cleanup-end, 0.0004520]
```

这之后，正常的 G1 周期就结束了——至少是垃圾的定位就完成了。清理阶段真正回收的内存数量很少，G1 到这个点为止真正做的事情是定位出哪些老的分区可回收垃圾最多（即图 6-7 中标记为 X 的分区）。

现在，G1 会执行一系列的混合式垃圾回收（mixed GC）。这些垃圾回收被称作“混合式”是因为它们不仅进行正常的新生代垃圾收集，同时也回收部分后台扫描线程标记的分区。混合式垃圾收集的效果如图 6-8 所示。

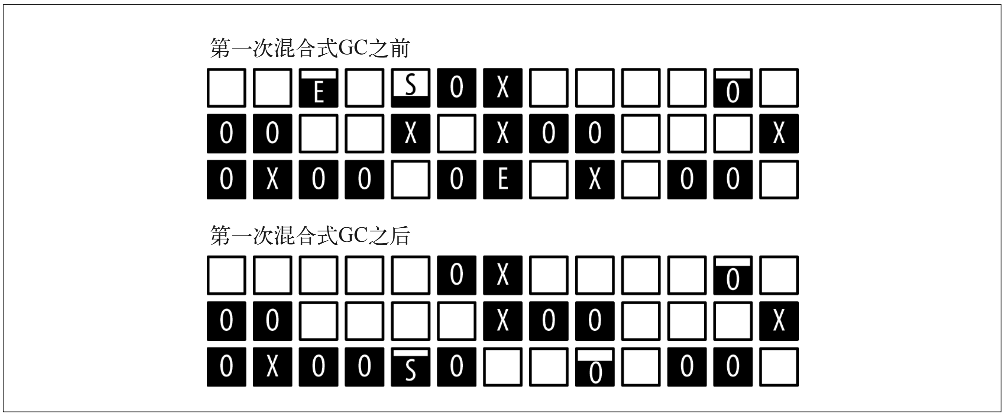


图 6-8：使用 G1 收集器进行的混合式 GC

同新生代垃圾收集通常的行为一样，G1 收集器已经清空了 Eden 空间，同时调整了 Survivor 空间的大小。此外，标记的两个分区也已经被回收。这些分区在之前的扫描中已经证实包含大量垃圾对象，因此绝大部分已经被释放。

这些分区中的活跃数据被移动到另一个分区（就像把活跃数据从新生代移动到老年代的分区）。这就是为什么 G1 收集器最终出现碎片化的堆的频率，跟 CMS 收集器比较起来要小得多的原因——随着 G1 垃圾的回收以这种方式移动对象，实际伴随着压缩。

关于混合式垃圾回收操作，请参考下面的日志：

```
79.826: [GC pause (mixed), 0.26161600 secs]
....
   [Eden: 1222M(1222M)->0B(1220M)
        Survivors: 142M->144M Heap: 3200M(4096M)->1964M(4096M)]
   [Times: user=1.01 sys=0.00, real=0.26 secs]
```

应注意，减少的整个堆的使用不仅仅是 Eden 空间移走的 1222 MB。这其中的差异看起来很小（只有 16 MB），但是同时还有部分 Survivor 空间的对象晋升到了永久代，除此之外，每次混合式垃圾回收只会清理部分目标老年代分区。接下来的讨论中，我们会看到确保混合式垃圾收集清理掉足够的内存对避免将来发生并发失效有多重要。

混合式垃圾回收周期会持续运行直到（几乎）所有标记的分区都被回收，这之后 G1 收集器会恢复常规的新生代垃圾回收周期。最终，G1 收集器会启动再一次的并发周期，决定哪些分区应该在下一次垃圾回收中释放。

同 CMS 收集器一样，有的时候你会在垃圾回收日志中观察到 Full GC，这些日志是一个信号，表明我们需要进一步调优（具体的方式很多，甚至很可能要分配更多的堆空间）才能提升应用程序的性能。主要有 4 种情况会触发这类的 Full GC，如下所列。

并发模式失效

G1 垃圾收集启动标记周期，但老年代在周期完成之前就被填满，在这种情况下，G1 收集器会放弃标记周期：

```
51.408: [GC concurrent-mark-start]
65.473: [Full GC 4095M->1395M(4096M), 6.1963770 secs]
 [Times: user=7.87 sys=0.00, real=6.20 secs]
71.669: [GC concurrent-mark-abort]
```

发生这种失败意味着堆的大小应该增加了，或者 G1 收集器的后台处理应该更早开始，或者是需要调整周期，让它运行得更快（譬如，增加后台处理的线程数）。

晋升失败

G1 收集器完成了标记阶段，开始启动混合式垃圾回收，清理老年代的分区，不过，老年代空间在垃圾回收释放出足够内存之前就会被耗尽。垃圾回收日志中，这种情况的现象通常是混合式 GC 之后紧接着一次 Full GC。

```
2226.224: [GC pause (mixed)
        2226.440: [SoftReference, 0 refs, 0.0000060 secs]
        2226.441: [WeakReference, 0 refs, 0.0000020 secs]
        2226.441: [FinalReference, 0 refs, 0.0000010 secs]
        2226.441: [PhantomReference, 0 refs, 0.0000010 secs]
        2226.441: [JNI Weak Reference, 0.0000030 secs]
                (to-space exhausted), 0.2390040 secs]
....
    [Eden: 0.0B(400.0M)->0.0B(400.0M)
        Survivors: 0.0B->0.0B Heap: 2006.4M(2048.0M)->2006.4M(2048.0M)]
    [Times: user=1.70 sys=0.04, real=0.26 secs]
2226.510: [Full GC (Allocation Failure)
        2227.519: [SoftReference, 4329 refs, 0.0005520 secs]
        2227.520: [WeakReference, 12646 refs, 0.0010510 secs]
        2227.521: [FinalReference, 7538 refs, 0.0005660 secs]
        2227.521: [PhantomReference, 168 refs, 0.0000120 secs]
        2227.521: [JNI Weak Reference, 0.0000020 secs]
                2006M->907M(2048M), 4.1615450 secs]
    [Times: user=6.76 sys=0.01, real=4.16 secs]
```

这种失败通常意味着混合式收集需要更迅速地完成垃圾收集；每次新生代垃圾收集需要处理更多老年代的分区。

疏散失败

进行新生代垃圾收集时，Survivor 空间和老年代中没有足够的空间容纳所有的幸存对象。这种情形在 GC 日志中通常被当成一种特别的新生代：

```
60.238: [GC pause (young) (to-space overflow), 0.41546900 secs]
```

这条日志表明堆已经几乎完全用尽或者碎片化了。G1 收集器会尝试修复这一失败，但是你可以预期，结果会更加恶化：G1 收集器会转而使用 Full GC。解决这个问题最简单的方式是增加堆的大小，除此之外，其他一些可能的解决方案会在第 159 页“高级调优”一节讨论。

巨型对象分配失败

使用 G1 收集器时，分配非常巨大对象的应用程序可能会遭遇另一种 Full GC；参见 6.4.2 节中“使用 G1 分配巨型对象”一节。目前为止没有工具可以很方便地专门诊断这种类型的失败，尤其是从标准垃圾收集日志中进行诊断。不过，如果发生了莫名其妙的

Full GC，其源头很可能是巨型对象分配导致的问题。

快速小结

1. G1 垃圾收集包括多个周期（以及并发周期内的阶段）。调优良好的 JVM 运行 G1 收集器时应该只经历新生代周期、混合式周期和并发 GC 周期。
2. G1 的并发阶段会产生少量的停顿。
3. 恰当的时候，我们需要对 G1 进行调优，才能避免 Full GC 周期发生。

G1垃圾收集器调优

G1 垃圾收集器调优的主要目标是避免发生并发模式失败或者疏散失败，一旦发生这些失败就会导致 Full GC。避免 Full GC 的技巧也适用于频繁发生的新生代垃圾收集，这些垃圾收集需要等待扫描根分区完成才能进行。

其次，调优可以使过程中的停顿时间最小化。下面所列的这些方法都能够避免发生 Full GC。

- 通过增加总的堆空间大小或者调整老年代、新生代之间的比例来增加老年代空间的大小。
- 增加后台线程的数目（假设我们有足够的 CPU 资源运行这些线程）。
- 以更高的频率进行 G1 的后台垃圾收集活动。
- 在混合式垃圾回收周期中完成更多的垃圾收集工作。

这里有很多的调优可以做，不过 G1 垃圾收集器调优的目标之一是尽量简单。为了达到这个目标，G1 收集器最主要的调优只通过一个标志进行：这个标志跟 Throughput 收集器的标志一致，也是 `-XX:MaxGCPauseMillis=N`。

使用 G1 垃圾收集器时，该标志有一个默认值：`200` 毫秒（这一点跟 Throughput 收集器有所不同）。如果 G1 收集器发生时空停顿（stop-the-world）的时长超过该值，G1 收集器就会尝试各种方式进行弥补——譬如调整新生代与老年代的比例，调整堆的大小，更早地启动后台处理，改变晋升阈值，或者是在混合式垃圾收集周期中处理更多或更少的老年代分区（这是最重要的方式）。

通常的取舍就发生在这里：如果减小参数值，为了达到停顿时间的目标，新生代的大小会相应减小，不过新生代垃圾收集的频率会更加频繁。除此之外，为了达到停顿时间的目标，混合式 GC 收集的老年代分区数也会减少，而这会增大并发模式失败发生的机会。

如果设置停顿时间目标无法避免 Full GC，我们可以进一步针对不同的方面逐一调优。对 G1 垃圾收集器而言，调整堆大小的方法与其他的垃圾收集算法并没有什么不同。

1. 调整G1垃圾收集的后台线程数

为了帮助 G1 赢得这场垃圾收集的比赛，可以尝试增加后台标记线程的数目（假设机器有足够的空闲 CPU 可以支撑这些线程的运行）。

调整 G1 垃圾收集线程的方法与调整 CMS 垃圾收集线程的方法类似：对于应用线程暂停运行的周期，可以使用 `ParallelGCThreads` 标志设置运行的线程数；对于并发运行阶段可以

使用 ConcGCThreads 标志设置运行线程数。不过，ConcGCThreads 标志的默认值在 G1 收集器中不同于 CMS 收集器。它的计算方法如下：

```
ConcGCThreads = (ParallelGCThreads + 2) / 4
```
[3]

这个算法依然是基于整数的；G1 收集器与 CMS 收集器的计算方法相差无几。

2. 调整G1垃圾收集器运行的频率

如果 G1 收集器更早地启动垃圾收集，也能赢得这场比赛。G1 垃圾收集周期通常在堆的占用达到参数 -XX:InitiatingHeapOccupancyPercent=*N* 设定的比率时启动，默认情况下该参数的值为 45。注意，跟 CMS 收集器不太一样，这个参数值的依据是整个堆的使用情况，不单是老年代的。

InitiatingHeapOccupancyPercent 的值是个常数，G1 收集器自身不会为了达到停顿时间目标而修改这个参数值。如果该参数设置得过高，应用程序会陷入 Full GC 的泥潭之中，因为并发阶段没有足够的时间在剩下的堆空间被填满之前完成垃圾收集。如果该值设定得过小，应用程序又会以超过实际需要的节奏进行大量的后台处理。我们在介绍 CMS 收集器时讨论过，必须要有能支撑后台处理的 CPU 周期，因此消耗额外的 CPU 就不那么重要。然而，这可能会带来非常严重的后果，因为并发阶段会出现越来越多的短暂应用线程的停顿。这些停顿会迅速累积起来，因此使用 G1 收集器时要避免频繁地进行后台清理。并发周期结束之后，检查下堆的大小，确保 InitiatingHeapOccupancyPercent 的值大于此时堆的大小。

3. 调整G1收集器的混合式垃圾收集周期

并发周期之后、老年代的标记分区回收完成之前，G1 收集器无法启动新的并发周期。因此，让 G1 收集器更早启动标记周期的另一个方法是在混合式垃圾回收周期中尽量处理更多的分区（如此一来最终的混合式 GC 周期就变少了）。

混合式垃圾收集要处理的工作量取决于三个因素。第一个因素是有多少分区被发现大部分是垃圾对象。目前没有标志能够直接调节这个因素：混合式垃圾收集中，如果分区的垃圾占用比达到 35%，这个分区就被标记为可以进行垃圾回收。（这个因素在将来的某个时刻可能也能调整，在开源的实验版本中已经有名为 -XX:G1MixedGCLiveThresholdPercent=*N* 的参数可以对其进行调整）。

第二个因素是 G1 垃圾收集回收分区时的最大混合式 GC 周期数，通过参数 -XX:G1MixedGCCountTarget=*N* 可以进行调节。这个参数的默认值为 8；减少该参数值可以帮助解决晋升失败的问题（代价是混合式 GC 周期的停顿时间会更长）。

另一方面，如果混合式 GC 的停顿时间过长，可以增大这个参数的值，减少每次混合式 GC 周期的工作量。不过调整之前我们需要确保增大值之后不会对下一次 G1 并发周期带来太大的延迟，否则可能会导致并发模式失败。

最后，第三个影响因素是 GC 停顿可忍受的最大时长（通过 MaxGCPauseMillis 参数设定）。MaxGCPauseMillis 标志设定的混合式周期时长是向上规整的，如果实际停顿时间在停顿最

注 3：CMS 收集器的 ConcGCThreads 计算公式为 ConcGCThreads = (3 + ParallelGCThreads) / 4。

大时长以内，G1 收集器能够收集超过八分之一标记的老年代分区（或者其他设定的值）。增大 `MaxGCPauseMillis` 能在每次混合式 GC 中收集更多的老年代分区，而这反过来又能帮助 G1 收集器在更早的时候启动并发周期。

快速小结

1. 作为 G1 收集器调优的第一步，首先应该设定一个合理的停顿时间作为目标。
2. 如果使用这个设置后，还是频繁发生 Full GC，并且堆的大小没有扩大的可能，这时就需要针对特定的失败采用特定的方法进行调优。
   a. 通过 `InitiatingHeapOccupancyPercent` 标志可以调整 G1 收集器，更频繁地启动后台垃圾收集线程。
   b. 如果有充足的 CPU 资源，可以考虑调整 `ConcGCThreads` 标志，增加垃圾收集线程数。
   c. 减小 `G1MixedGCCountTarget` 参数可以避免晋升失败。

6.4　高级调优

这一节我们会讨论一些比较少见的性能问题调优。虽然这些场景不常碰到，但是垃圾收集算法涉及的很多底层细节会在这一节中详细讨论。

6.4.1　晋升及Survivor空间

新生代垃圾收集时，有的对象可能还处于活跃期。这些对象中，有些是刚创建的新对象，这些对象还会存活相当长的一段时间，还有一些只有短暂的生命周期。我们以第 5 章中讨论过的计算 `BigDecimal` 的循环为例。如果 JVM 在循环的中段启动垃圾回收，这些超短寿（very-short-lived）的 `BigDecimal` 对象面临的局面就变得非常尴尬：它们刚被创建，因此不能被回收释放；但是它们的生命周期又非常短，无法满足晋升到老年代的条件。

这就是新生代被划分成一个 Eden 空间和两个 Survivor 空间的原因。这种布局让对象在新生代内有更多的机会被回收，不再局限于只能晋升到老年代（最终填满老年代）。

新生代垃圾收集时，如果 JVM 发现对象还十分活跃，会首先尝试将其移动到 Survivor 空间，而不是直接移动到老年代。首次新生代垃圾收集时，对象被从 Eden 空间移动到 Survivor 空间 0。紧接着的下一次垃圾收集中，活跃对象会从 Survivor 空间 0 和 Eden 空间移动到 Survivor 空间 1。这之后，Eden 空间和 Survivor 空间 0 被完全清空。下一次的垃圾回收会将活跃对象从 Survivor 空间 1 和 Eden 空间移回 Survivor 空间 0，如此反复。（Survivor 空间也被称为"To"空间和"From"空间；每次回收，对象由"From"空间移出，移入到"To"空间。"From"和"To"只是简单地表示两个 Survivor 空间之间的指向，每次垃圾回收时，方向都会互换。）

显而易见，这种状况不会一直持续下去，否则没有任何对象会进入老年代。两种情况下，对象会被移动到老年代。第一，Survivor 空间的大小实在太小。新生代垃圾收集时，如果目标 Survivor 空间被填满，Eden 空间剩下的活跃对象会直接进入老年代。第二，对象在

Survivor 空间中经历的 GC 周期数有个上限，超过这个上限的对象也会被移动到老年代。这个上限值被称为晋升阈值（Tenuring Threshold）。

这些影响垃圾收集的因素都有对应的调优标志。Survivor 空间是新生代空间的一部分，跟堆内的其他区域一样，JVM 可以对它进行动态的调节。Survivor 空间的初始大小由 `-XX:InitialSurvivorRatio=N` 标志决定。这个参数值在下面的这个公式中使用：

```
survivor_space_size = new_size / (initial_survivor_ratio + 2)
```

初始 Survivor 空间的占用比率（initial_survivor_ratio）默认为 8，由此我们可以计算出每个 Survivor 空间会占用大约 10% 的新生代空间。

JVM 可以增大 Survivor 空间的大小直到其最大上限，这个上限可以通过 `-XX:MinSurvivorRatio=N` 参数设置。`MinSurvivorRatio` 标志在下面这个公式中使用：

```
maximum_survivor_space_size = new_size / (min_survivor_ratio + 2)
```

这个参数值默认为 3，意味着 Survivor 空间的最大值为新生代空间的 20%。再次提醒，这个参数值是个分母，分母值最小时，Survivor 空间的容量最大。这样说起来，这个参数的名字的确有些不直观。

为了保持 Survivor 空间的大小为某个固定值，我们可以使用 `SurvivorRatio` 参数，将其设定为期望的值，同时关闭 `UseAdaptiveSizePolicy` 标志（然而，我们需要注意一点，即关闭自适应大小调整会同时影响新生代和老年代）。

JVM 依据垃圾回收之后 Survivor 空间的占用情况判断是否需要增加或者减少 Survivor 空间的大小（由定义的比率决定）。默认情况下，Survivor 空间调整之后要能保证垃圾回收之后有 50% 的空间是空闲的。通过标志 `-XX:TargetSurvivorRatio=N` 可以设置这个值。

最后，还有一个问题，即对象在移动到老年代之前，需要在 Survivor 空间之间来回移动多少个 GC 周期。这个问题取决于晋升阈值的设定。JVM 会持续地计算，寻找它认为最合适的晋升阈值。通过 `-XX:InitialTenuringThreshold=N` 标志可以设置初始的晋升阈值（对于 Throughput 收集器和 G1 收集器，默认值是 7，对于 CMS 收集器默认值为 6）。JVM 最终会在 1 和最大晋升阈值（由 `-XX:MaxTenuringThreshold=N` 标志设定）之间选择一个合适的值。对于 Throughput 收集器和 G1 收集器，默认的最大晋升阈值为 15，对 CMS 收集器，最大的晋升阈值为 6。

两种窘境：一直“晋升”与从不“晋升”

晋升阈值总是在 1 到 `MaxTenuringThreshold` 之间取值。即使 JVM 启动时将初始晋升阈值设置为最大值，这个参数也不一定会一直保持，JVM 可能会在某个时刻减小这个阈值。

使用两个标志可以通过极端的方式避免出现这种情况。如果你确切地知道新生代垃圾收集存活下来的对象在之后很长的一段时间内都会存在，可以使用 `-XX:+AlwaysTenure` 标志（默认值为 `false`），开启这个标志的效果与将 `MaxTenuringThreshold` 设置为 0 的效果在本质上是一样的。然而，只有非常罕见的情况才需要开启这个标志，启用之后对象会直接晋升到老年代，不会再存放于 Survivor 空间。

> 第二个标志是 -XX:+NeverTenure（默认值也是 false）。这个标志有两方面的影响：设置参数后 JVM 会认为初始晋升阈值和最大晋升阈值都无限大；一旦设置了该参数，JVM 就不再调整晋升阈值，也不会将其降低。换句话说，开启 -XX:+NeverTenure 标志后只要 Survivor 空间有容量，就不会有对象被晋升到老年代。

我们已经学习了很多的参数，那么什么情况下应该使用哪些参数呢？观察晋升的统计信息能够帮助我们更好地做出决定。使用 -XX:+PrintTenuringDistribution 标志可以在 GC 日志中增加这部分信息（默认情况下，-XX:+PrintTenuringDistribution 的值为 false）。

查看 GC 日志时，最重要的是观察在 Minor GC 中是否存在由于 Survivor 空间过小，对象直接晋升到老年代的情况。我们要尽量避免发生这种情况：如果大量的短期对象最终填满老年代，会导致频繁的 Full GC。

使用 Throughput 收集器时，判断发生了这种情况的唯一线索是下面这几行 GC 日志：

```
Desired survivor size 39059456 bytes, new threshold 1 (max 15)
         [PSYoungGen: 657856K->35712K(660864K)]
         1659879K->1073807K(2059008K), 0.0950040 secs]
         [Times: user=0.32 sys=0.00, real=0.09 secs]
```

从日志中我们看到，例子中一个 Survivor 空间期望的大小是 39 MB，新生代的总大小为 660 MB：JVM 据此计算出两个 Survivor 空间大约要占用 11% 的新生代空间。不过这又留给我们一个问题：这部分空间是否已经足够大，是否能避免发生新生代到老年代的溢出。垃圾收集日志无法直接回答这个问题，但是从 JVM 将晋升阈值调整到 1 这个事实，我们可以判断 JVM 会直接晋升大部分对象到老年代，并据此将晋升阈值减小到 1。这个应用极可能在 Survivor 空间还未完全填满时就将对象直接晋升到老年代。

使用 G1 收集器或 CMS 收集器时，我们可以从垃圾收集日志中获取更多的信息：

```
Desired survivor size 35782656 bytes, new threshold 2 (max 6)
- age   1:   33291392 bytes,   33291392 total
- age   2:    4098176 bytes,   37389568 total
```

期望的 Survivor 空间与上一个例子很相似，大约是 35 MB，但是我们能看到更多信息，包括 Survivor 空间中所有对象的大小。由于需要晋升 37 MB 的数据，Survivor 空间的确会发生溢出。

这种情况能否通过调优改善取决于应用程序自身的特性。如果对象的生命周期很长，跨越多个垃圾收集周期，无论怎样调整它们最终都会移动到老年代，在这种情况下，调整 Survivor 空间和晋升阈值不会有太大的帮助。但是，如果对象经过几个 GC 周期就会被回收，合理安排 Survivor 空间更高效地加以利用，能够提升一定的程序性能。

如果（通过减小生存比率的方式）增大 Survivor 空间的大小，内存由新生代的 Eden 空间划分到 Survivor 空间。不过对象的分配都发生在 Eden 空间，这意味着在 Minor GC 之前能分配的对象数目会更少。因此，我们不推荐采用这种方式。

另一种可能是增大新生代的大小。采用这种方式的效果可能事与愿违：虽然对象晋升到老年代的频率降低了，但是老年代空间变得更小，应用程序可能会更频繁地发生 Full GC。

如果堆的大小可以同时增加，那么新生代和老年代都能获得更多的内存，这是最好的解决方案。推荐的流程是增大堆的大小（或者至少增大新生代），同时减小存活率。采用这种方法 Survivor 空间增大的值会比 Eden 空间的增长更大。应用程序最终的新生代垃圾收集次数与调节之前基本持平。不过 Full FC 的次数会更少，因为晋升到老年代的对象数更少了（再次重申，这种调优适用的应用程序，其大多数对象在几个 GC 周期之后就不再存活）。

如果 Survivor 空间经过调整后不再发生溢出，对象只有在经历的 GC 周期数达到 `MaxTenuringThreshold` 的设定值时才会晋升到老年代。我们可以增大 `MaxTenuringThreshold` 值，让对象在 Survivor 空间中停留更多的周期。但是，我们也要注意，晋升阈值增大，对象在 Survivor 空间停留的时间越长，将来的新生代收集中，Survivor 空闲空间就会越少：越有可能发生 Survivor 空间溢出，对象再次被直接晋升到老年代。

快速小结

1. 设计 Survivor 空间的初衷是为了让对象（尤其是已经分配的对象）在新生代停留更多的 GC 周期。这个设计增大了对象晋升到老年代之前被回收释放的几率。
2. 如果 Survivor 空间过小，对象会直接晋升到老年代，从而触发更多的老年代 GC。
3. 解决这个问题的最好方法是增大堆的大小（或者至少增大新生代），让 JVM 来处理 Survivor 空间的回收。
4. 有的情况下，我们需要避免对象晋升到老年代，调整晋升阈值或者 Survivor 空间的大小可以避免对象晋升到老年代。

6.4.2　分配大对象

这一节会详细介绍 JVM 是如何分配对象的。这是一些非常有趣的背景知识，了解这些对于调优需要频繁创建大量大型对象的应用尤其重要。这一节的上下文中，“大型”是一个相对的概念；正如我们后面会看到的，它取决于 JVM 内的“线程本地分配缓冲区”（Thread Local Allocation Buffer，TLAB）。

TLAB 的大小是各种垃圾收集算法进行垃圾收集时都要考虑的因素，除此之外，G1 收集器对超大型对象还有一些额外的考量（再次重申，大型是个相对的术语，对于 2 GB 的堆，对象大小如果超过 512 MB 就算大型对象）。G1 收集器在收集巨型对象时效果非常显著，对线程本地分配缓冲区的大小调整（使用一般的垃圾收集器，为了处理较大对象）则并不常见；不过，G1 收集器的分区大小调整（使用 G1 收集器时，为了处理巨型对象）就比较司空见惯了。

1. TLAB

第 5 章讨论了对象是如何在 Eden 空间中分配的；Eden 空间让更快地进行对象分配成为可能（尤其是对于分配之后又被迅速回收的对象）。

结果表明，Eden 空间中对象分配速度更快的原因是每个线程都有一个固定的分区用于分

配对象，即一个 TLAB。对象在一个共享的空间中分配，我们需要采用一些同步机制来管理空间内的空闲空间指针。每个线程有固定的分配区域，分配对象时，线程之间不需要进行任何的同步。（这是利用线程本地变量避免锁争用（详细内容请参考第 9 章）的一个变种。）

通常，TLAB 的使用对程序员和终端用户而言是完全透明的：默认情况下 TLAB 就是开启的，JVM 管理着它们的大小及如何使用。我们需要意识到的最重要的事是 TLAB 都不大，因此大型对象无法在 TLAB 内进行分配。大型对象必须直接从堆上分配，由于需要同步，这会消耗额外的时间。

一旦 TLAB 空间用尽，特定大小的对象就无法再继续分配。这时，JVM 可以有不同的选择。一个选项是回收这块 TLAB，为该线程分配一块新的 TLAB。由于 TLAB 只是 Eden 空间中的一个区段，下一次新生代垃圾收集时这块 TLAB 整个都会被回收，并在之后的空间分配中重用。除此之外，JVM 还可以直接在堆上分配对象，保留当前的 TLAB 不动（至少在线程分配新的对象到 TLAB 之前保持不变）。假设发生下面这种情况，TLAB 的大小为 100 KB，其中 75 KB 都已经被占用。这时来了个新的空间分配请求，需要分配 30 KB 的空间，我们可以回收整个 TLAB，这种方式会浪费 25 KB 的 Eden 空间。或者直接在堆上分配这个对象，如果下一次的对象分配空间要求小于等于 25 KB，线程还可以将 TLAB 的空闲空间分配给这些对象。

JVM 提供了各种参数可以控制这些行为（本节后面会详细讨论这些参数），但这一切都取决于 TLAB 大小。默认情况下，TLAB 的大小由三个因素决定：应用程序的线程数、Eden 空间的大小以及线程的分配率。

因此两类的应用程序会受益于 TLAB 参数的调整：需要分配大量巨型对象的应用程序，以及相对于 Eden 空间的大小而言，应用程序线程数量过多的应用。默认情况下，TLAB 就是开启的；使用 `-XX:-UseTLAB` 可以关闭 TLAB，不过考虑到 TLAB 带来的性能提升，关闭这个功能不是个明智的决定。

由于 TLAB 空间大小的计算在一定程度上基于线程的分配率，我们不大可能准确预测应用程序的 TLAB 大小。我们能做的是监控 TLAB 的分配情况，看是否有任何对象的分配发生在 TLAB 之外。如果发现大量的对象分配发生在 TLAB 之外，我们有两种选择：减小分配对象的大小，或者调整 TLAB 的参数。

与其他的工具比较起来，JFR 在 TLAB 分配的监控方面要强大得多。图 6-9 展示了使用 JFR 记录 TLAB 分配的示例截屏。

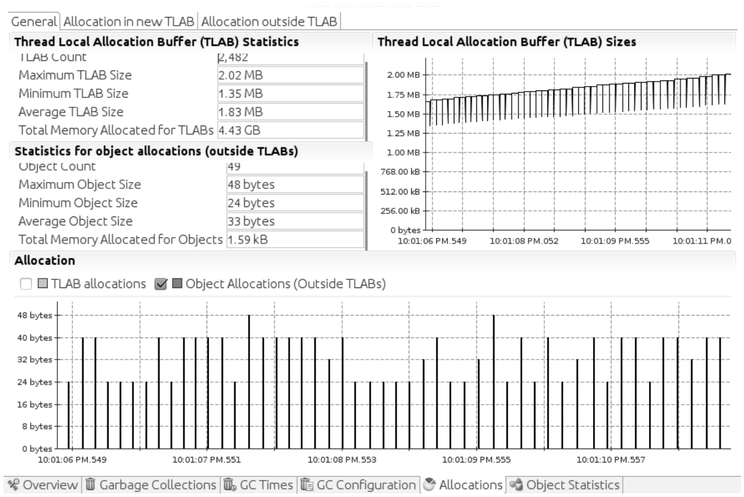


图 6-9：JFR 中的 TLAB 视图

在这段 5 秒钟的记录中，49 个对象分配发生在 TLAB 之外；这些对象的最大值为 48 字节。由 TLAB 的最小值为 1.35 MB 我们得知，这些对象被分配到堆上的原因是空间分配时 TLAB 已经耗尽：它们不是由于对象大小的原因被直接分配到了堆上。这种情况通常发生在新生代垃圾回收之前（由于 Eden 空间耗尽，而 TLAB 是 Eden 空间切出来的一部分）。

这段时间内，对象分配的总大小为 1.59 KB，这个例子中无论是分配的数量，还是分配的大小都不是问题。总会有一些对象在 TLAB 之外分配，尤其是当 Eden 空间的使用接近新生代收集的边缘时。我们可以对比这个例子跟图 6-10 中的情况，图 6-10 中大量的分配发生在 TLAB 之外。

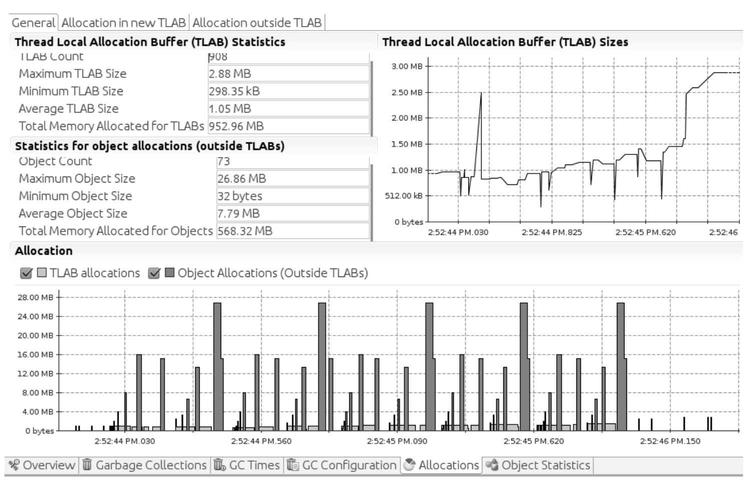


图 6-10：大量的分配发生在 TLAB 之外

这段记录中，TLAB 中内存分配的总数是 952.96 MB，TLAB 之外对象分配使用的总内存数是 568.32 MB。对于这个例子，无论是改变应用程序，使用更小的对象，还是调整 JVM 将更大的 TLAB 分配给这些对象，都能取得不错的效果；我们甚至可以得到这些对象分配时的堆栈。如果 TLAB 分配中发生了问题，使用 JFR 能迅速地定位出到底什么地方出现了问题。

对于开源版本的 JVM（不附带 JFR），要监控 TLAB 的分配情况，最好的途径就是在命令行中添加 `-XX:+PrintTLAB` 标志。这样，每次新生代垃圾收集时，GC 日志中就同时包含了两种类型的信息：每个线程都有一行描述该线程的 TLAB 使用情况，以及一行摘要信息，描述 JVM 整体的 TLAB 使用情况。

每个线程一行的日志如下所示：

```
TLAB: gc thread: 0x00007f3c10b8f800 [id: 18519] desired_size: 221KB
    slow allocs: 8  refill waste: 3536B alloc: 0.01613    11058KB
    refills: 73 waste  0.1% gc: 10368B slow: 2112B fast: 0B
```

输出中的 `gc` 表明这一行日志是在垃圾回收时输出的；线程自身是一个常规的应用线程。线程的 TLAB 大小是 221 KB。从上次新生代收集开始，已经有八个对象在堆上分配（`slow allocs`）；占线程分配对象总量的 1.6%（0.01613），总计大小为 11 058 KB。TLAB 空间的 0.1% 被“浪费”了，主要的源头是三件事：并发垃圾收集启动时 TLAB 中的 10 336 字节空间被回收释放，其他（释放的）TLAB 中 2112 字节被释放，以及“快速”分配器分配的空间大小为“0”字节。

每个线程的 TLAB 数据输出后，JVM 还会输出一条概略日志，如下所示：

```
TLAB totals: thrds: 66  refills: 3234 max: 105
        slow allocs: 406 max 14 waste:  1.1% gc: 7519856B
        max: 211464B slow: 120016B max: 4808B fast: 0B max: 0B
```

在这个例子中，从上次新生代垃圾收集起，66 个线程进行了各种形式的内存分配。这些线程总共填充了 TLAB 3234 次，最活跃的线程填充了它的 TLAB 105 次。总共在堆上分配了 406 次对象（一个线程最多分配了 14 次对象），并且 TLAB 中有 1.1% 的空闲空间由于 TLAB 空间的释放被浪费。

在每个线程的日志中，如果发现线程有大量的对象分配发生在 TLAB 之外，就应该考虑对 TLAB 进行调整了。

2. 调整TLAB的大小

对于花费大量时间在 TLAB 之外分配对象的应用程序，将分配移动到 TLAB 之内能有效提升应用程序的性能。如果只有极少数对象的分配发生在 TLAB 之外，提升性能最好的方案是修改应用程序。

如果不可能变更应用程序代码，你还可以尝试通过调整 TLAB 的大小来适配应用程序的需要。由于 TLAB 的大小基于 Eden 空间，通过参数调整（增大）Eden 空间会自动增大 TLAB 的大小。

使用 `-XX:TLABSize=`*N* 标志可以显式地指定 TLAB 的大小（默认为 0，表示使用前面介绍的

方法动态计算得出)。这个标志只能设置 TLAB 的初始大小;为了避免在每次 GC 时都调整 TLAB 的大小,可以使用 -XX:-ResizeTLAB 标志(大多数的平台上,这个参数的默认值都是 true)。这是通过调整 TLAB,充分提升对象分配性能最简单的方法(坦率地说,通常这也是最有效的方法)。

一个新的对象无法适配到当前的 TLAB 中(但是可以容纳于一个新的、空闲的 TLAB 中)时,JVM 就需要做一些抉择:到底是在堆上分配这个对象,还是要回收当前的 TLAB,重新分配一个新的 TLAB 来完成这次对象分配,这个决策取决于几个参数。TLAB 日志的输出中,refill waste 的值代表了决策的当前阈值:如果 TLAB 无法容纳新对象的大小超过这个阈值,那么就会在堆上分配新的对象。如果有问题的对象的大小比这个阈值小,就回收老的 TLAB 空间。

这个值是动态计算得出的,但是默认的初始值是 TLAB 大小的 1%,或者是由参数 -XX:TLABWasteTargetPercent=*N* 特别设定的值。每当发生堆上的分配,这个值就增大一笔,增量值由参数 -XX:TLABWasteIncrement=*N* 设定(默认值为 4)。这种设计能够避免线程达到 TLAB 空间占用的阈值,从而持续地在堆上分配对象:随着目标百分比(Target Percentage)的增大,TLAB 空间被回收的几率也在增加。调整 TLABWasteTargetPercent 参数的结果往往同时伴随着 TLAB 空间大小的调整,所以,虽然可以调整这个参数,但是效果往往不那么确定。

最终,TLAB 空间调整生效时,其容量的最小值可以使用 -XX:MinTLABSize=*N* 参数设置(默认为 2 KB)。TLAB 空间的最大容量略小于 1 GB(使用整型数组可以用到 TLAB 空间的最大上限,由于对象对齐的原因,最大上限会向下圆整),并且不能修改。

快速小结

对需要分配大量大型对象的应用,TLAB 空间的调整就变得必不可少(不过,通常情况下,我们更推荐在应用程序中使用小型对象的做法)。

3. 巨型对象(Humongous Objects)

对 TLAB 空间中无法分配的对象,JVM 会尽量尝试在 Eden 空间中进行分配。如果 Eden 空间无法容纳该对象,就只能在老年代中分配空间。而这种内存布局打乱了该对象正常的垃圾回收周期,如果它是一个短期存在的对象,还会对垃圾收集造成负面的影响。对于这种情况,除非修改应用程序,放弃使用那些短期存在的巨型对象,否则别无它法。

G1 收集器使用不同的方法处理巨型对象,不过如果对象的大小超过了 G1 收集器的分区,这些对象也会被分配到老年代。因此,对于使用大量巨型对象的应用程序,即使使用 G1 收集器还是需要特别的调优才能弥补这部分的性能损失。

4. G1分区的大小

G1 收集器将堆划分成了一定数量的分区,每个分区的大小都是固定的。分区的大小不是动态变化的,具体的值是启动时,依据堆大小的最小值(即 Xms 的值)得出的。分区大小的最小值是 1 MB。如果堆的最小值超过 2 GB,分区的大小会依据下面的公式计算得出(使用基数为 2,取 log 的算法):

```
分区大小 = 1 << log(初始堆的大小 / 2048);
```

简言之，初始划分堆时，分区的大小是 2 的最小的 N 次幂，使其结果最接近于 2048 个分区。这里还有一些最小、最大值的限制；分区的大小最小是 1 MB，最大不能超过 32 MB。表 6-3 列出了所有的可能性。

表6-3：G1收集器的默认分区大小

堆的大小	默认G1分区的大小
小于 4 GB	1 MB
介于 4 GB 到 8 GB 之间	2 MB
介于 8 GB 到 16 GB 之间	4 MB
介于 16 GB 到 32 GB 之间	8 MB
介于 32 GB 到 64 GB 之间	16 MB
大于 64 GB	32 MB

G1 分区的大小可以通过 `-XX:G1HeapRegionSize=N` 标志设置（正常情况下，默认值是 0，意味着使用刚才描述的动态算法计算）。设定的参数值应该是 2 的幂（譬如：1 MB 或者 2 MB）；否则，这个值会向下圆整到最接近 2 的幂。

G1 收集的分区调整及大堆

通常情况下，G1 收集器分区的大小调整只有在处理巨型对象分配时才需要进行，但是也有一种例外的情形。

如果应用程序设定了一个非常大的堆区间，譬如 `-Xms2G -Xmx32G`，这种情况下分区的大小是 1 MB。当堆充分扩张时，G1 收集器的分区数可以高达 32 000 个。这是一个数量巨大的待处理分区，G1 收集算法最初的设计并没有针对这样大量的分区，它期望的分区数是 2048 个左右。这个例子中，增大 G1 收集器分区的大小能提高 G1 垃圾收集的效率；我们需要依据堆的大小选择合适的分区大小，让分区的数量尽量接近 2048 个。

5. 使用G1收集器分配巨型对象

如果 G1 收集器的分区大小是 1 MB，应用程序分配了个 2 百万字节的数组，这个数组是没有办法在一个 G1 分区中存放的。但是这些巨型对象又必须被保存在连续的 G1 分区内。如果 G1 分区的大小是 1 MB，程序分配了个 3.1 MB 的数组，G1 收集器必须在老年代内找到 4 个连续的分区才能完成这次分配工作（最后一个分区的剩余部分会保持空闲，导致 0.9 MB 的空间浪费）。这种做法成功地打败了 G1 收集器传统的收集方式，即压缩，它能够依据分区的满溢程度自主地选择回收哪些分区。通常，为了找到连续的分区，G1 收集器还不得不启动 Full GC。

由于巨型对象直接在老年代中分配，它们不会被新生代垃圾收集所回收。因此，如果短寿型对象采用这种方式分配，收集器的分代机制就不再生效。巨型对象只能在 G1 收集器的并发周期中回收。好消息是，巨型对象的回收会更迅速，因为它是所在分区唯一的对象。巨型对象会在并发周期中的清理阶段（而不是混合式 GC 阶段）被回收释放。

增大 G1 分区的大小，让其能够在一个分区内分配应用需要的所有对象能够提升 G1 收集的效率。为了判断应用的 Full GC 是否源于巨型对象的分配，我们需要开启自适应大小调整（Adaptive Size Policy）GC 的日志记录。应用程序分配巨型对象时，G1 收集器首先会尝试启动一个并发周期。

```
5.349: [G1Ergonomics (Concurrent Cycles) request concurrent cycle initiation,
    reason: occupancy higher than threshold, occupancy: 483393536 bytes,
    allocation request: 524304 bytes, threshold: 483183810 bytes (45.00 %),
    source: concurrent humongous allocation]
  ...
5.350: [GC pause (young) (initial-mark) 0.349: [G1Ergonomics
    (CSet Construction) start choosing CSet, _pending_cards:
    1624, predicted base time: 19.74 ms, remaining time: 180.26 ms,
    target pause time: 200.00 ms]
```

这行日志表明发生了巨型对象分配，触发了并发 G1 周期。在这个例子中，对象分配成功，没有堆垃圾收集的其他方面造成影响（G1 收集器刚好找到了需要的连续分区）。

如果 G1 收集器没有找到连续的空闲分区，就会启动一次 Full GC：

```
25.270: [G1Ergonomics (Heap Sizing) attempt heap expansion,
    reason: allocation request failed, allocation request: 48 bytes]
25.270: [G1Ergonomics (Heap Sizing) expand the heap,
    requested expansion amount: 1048576 bytes,
    attempted expansion amount: 1048576 bytes]
25.270: [G1Ergonomics (Heap Sizing) did not expand the heap,
    reason: heap expansion operation failed]
25.270: [Full GC 1535M->1521M(3072M), 1.0358230 secs]
      [Eden: 0.0B(153.0M)->0.0B(153.0M)
       Survivors: 0.0B->0.0B Heap: 1535.9M(3072.0M)->1521.3M(3072.0M)]
      [Times: user=5.24 sys=0.00, real=1.04 secs]
```

由于堆无法为适配新的巨型对象而进行扩展，因此为了给分配请求提供连续的分区，G1 收集器只能进行 Full GC。在这种情况下，一旦发生问题，即便是开启 `PrintAdaptiveSizePolicy` 标志也无法提供更多的垃圾回收日志，标准的 G1 垃圾收集日志也无法提供足够的信息，以诊断问题的根源。

为了避免发生这种 Full GC，首先是要确定导致问题的巨型对象大小（本例中，从垃圾收集日志中定位的对象大小是 524 304 字节）。接下来，需要判断是否有办法减少程序中这些对象的大小。下下策才是针对这些对象，调整 JVM。如果无法减少对象的大小，就需要计算容纳这些对象所需要的分区大小。如果对象占用的空间达到分区容量的 50% 以上，G1 收集器就认为这是个巨型对象。因此，这个例子中，如果被质疑的对象大小为 524 304 字节，G1 分区的大小至少应该是 1.1 MB。由于 G1 收集算法中，分区的大小总是 2 的幂，所以 G1 分区的大小应该为 2 MB，才能保持在标准的 G1 分区中完成这些对象的分配。

快速小结

1. G1 分区的大小是 2 的幂，最小值为 1 MB。
2. 如果堆的初始大小跟最大值相差很大，这种堆会有大量的 G1 分区，在这种情况下，应该增大 G1 分区的大小。

3. 如果要分配的对象大小超过了 G1 收集器分区容量的一半，对于这种应用程序，我们应该增大 G1 分区的容量，让 G1 分区能更好地适配这些对象。遵循这个原则，应用程序分配对象的大小至少应是 512 KB（因为 G1 分区的最小值为 1 MB）。

6.4.3 AggressiveHeap标志

`AggressiveHeap` 在非常早期的 Java 版本中就已经引入（默认为 `false`）。那个时候，配置了大量内存的大型服务器也只运行单一的 JVM，使用 `AggressiveHeap` 标志能更方便地设置各种命令行参数。这个标志只适用于 64 位的 JVM。

虽然这个标志已经更换了很多个版本，到目前依然存在，但是我们不推荐使用该标志（虽然这个标志暂时还没有被官方正式弃用）。这个标志的问题在于它隐藏了很多实际采用的调优工作，让我们很难了解 JVM 实际运行时的设置。它的一些参数值设置是依据运行 JVM 中物理机的情况动态调整的，因此，开启这个标志有时对性能的影响是负面的。我经常看到使用这个标志的命令行，在之后实际运行时标志被动态修改的情况。（对于历史数据，这个标志可能会有效：不过，之后的运行时命令行标志值会改写之前设定的值。这种行为是不确定的。）

表 6-4 列出了开启 `AggressiveHeap` 标志后会被自动调整的参数。

表6-4：启用AggressiveHeap打开的调优标志

标志	值
`Xmx`	所有内存的 1/2；或者所有的内存；或者 160 MB
`Xms`	与 `Xmx` 相同
`NewSize`	`Xmx` 设定值的 3/8
`UseLargePages`	`true`
`ResizeTLAB`	`false`
`TLABSize`	256 KB
`UseParallelGC`	`true`
`ParallelGCThreads`	与当前的默认值相同
`YoungPLABSize`	256 KB（默认为 4 KB）
`OldPLABSize`	8 KB（默认为 1 KB）
`CompilationPolicyChoice`	0（当前默认值）
`ThresholdTolerance`	100（默认为 10）
`ScavengeBeforeFullGC`	`false`（默认为 `true`）
`BindGCTaskThreadsToCPUs`	`true`（默认为 `false`）

最后六个标志相当晦涩，本书其他地方都没讨论过。简单地说，它们主要负责以下所列这些方面。

PLAB 的大小

PLAB 的全称是晋升本地分配缓冲（Promotion-Local Allocation Buffer），是垃圾回收清理代数据时基于线程分配的分区。每个线程都能晋升对象到自己的 PLAB 中，避免了相

互同步的需求（工作原理与 TLAB 很相似）。

编译策略

JVM 发布时配备了多种 JIT 编译算法。当前默认的算法也曾经一度是实验性的，但是随着时间的打磨，它们不断完善成熟，变成了现在推荐的策略。

关闭 Full GC 之前的新生代垃圾收集

将 `ScavengeBeforeFullGC` 标志设置为 `false` 意味着 Full GC 发生时，JVM 不会对 Full GC 之前的新生代垃圾进行收集。通常这不是件好事儿，因为这意味着新生代中的垃圾对象（针对的是能够进行收集的垃圾对象）会妨碍老年代的对象被回收。显然，这些设置总有其存在的理由，不论过去或者现在（至少是在进行某些基准测试时可能有需要），但是，我们通常的建议是不要修改该标志。

将 GC 线程绑定到特定的 CPU

对上述列表中的最后一个标志进行设置意味着将各个并行垃圾收集线程分别绑定到特定的 CPU（通过操作系统相关的调用）。有些非常极端的情况，譬如垃圾收集线程是运行于该机器上的唯一任务，同时堆的容量又足够大时，使用该标志是有意义的。通常情况下，最好允许垃圾收集线程在所有可用的 CPU 上运行。

跟所有的调优一样，各个方法虽有不同，不过最终都是殊途同归，如果经过认真的测试，发现使用 `AggressiveHeap` 标志的确改善了程序性能，那就尽量使用该标志。唯一要留意的是，我们需要了解标志的背后到底发生了什么样的调优。应意识到，一旦 JVM 进行了升级，使用该标志带来的所有性能提升都需要重新评估。

快速小结

1. `AggressiveHeap` 是个历史悠久的调优标志，设计初衷是为了在强大的机器上运行单一 JVM 时调整堆的各种参数。
2. 这个标志设定的值并没有随着 JVM 技术的发展同步调整，因此它的有效性从长远来看是值得质疑的（虽然到目前为止，这个标志还常常被使用）。

6.4.4 全盘掌控堆空间的大小

本书 5.2.1 节“调整堆的大小”围绕堆的初始最小空间和最大空间的默认值展开了讨论。这些值的设定取决于机器的内存以及 JVM 已经使用了多少内存，并且之前介绍的设置还存在不少极端情况。如果你对堆的默认空间大小是如何计算出来的感到好奇，读完这一节也许能豁然开朗。本节会将堆空间计算的内幕一一展现在你眼前。我们要介绍的内容包括非常底层的调优标志；在有的情况下，直接调整堆空间的计算方法（而不只是设置堆的大小）可能更直观。譬如下面这个例子：你希望采用通用的堆大小（而不是针对每个虚拟机单独调优的堆）来运行多个 JVM。本节大多数部分的目标还是介绍堆的各个默认选择的前世今生，让大家能有个全局性的了解。

堆的默认大小依据机器的内存配置确定，不过也可以通过参数 `-XX:MaxRAM=`*`N`* 设置。通常情况下，这个值是由 JVM 检测机器的物理内存计算得出。不过，JVM 同时设置了一些限

制，譬如对于 32 位的 client 编译器，`MaxRam` 最大只能是 1 GB，对于 32 位的 server 编译器，`MaxRam` 可达 4 GB，如果是 64 位编译器，上限最大可以达到 128 GB。堆的最大容量是 `MaxRAM` 值的四分之一。这就是为什么堆的默认大小在不同的机器上会有不同的原因：如果机器的物理内存比 `MaxRAM` 的值小，默认堆的大小就是物理内存的 1/4。但是，相反的规则并不适用，即使机器配置了数百 GB 的内存，JVM 能使用的最大堆容量也不会超过默认值 32 GB，即 128 GB 的 1/4。

默认最大堆的计算实际采用下面的公式：

```
Default Xmx = MaxRAM / MaxRAMFraction
```

因此，默认最大堆的大小也可以通过 `-XX:MaxRAMFraction=N` 标志值进行调整，`MaxRAMFraction` 的默认值为 4。最后，为了让堆的默认值调整更加完备，JVM 还提供了另一个参数调整最大堆的默认值，这个参数是 `-XX:ErgoHeapSizeLimit=N`。该参数默认值为 0（表示忽略该标志）；否则，如果设置的限制值比 `MaxRAM/MaxRAMFraction` 还小，就使用该参数指定的值。

另一方面，如果机器配置的物理内存非常少，JVM 还要确保预留足够的内存给操作系统使用。这就是为什么在内存只有 192 MB 的机器上，JVM 会限制最大堆的大小为 96 MB 甚至更少。这个值的计算是基于 `-XX:MinRAMFraction=N` 参数，默认值为 2。

```
if ((96 MB * MinRAMFraction) > Physical Memory) {
    Default Xmx = Physical Memory / MinRAMFraction;
}
```

计算堆的初始大小与此类似，不过相对简单一些，影响因素更少。堆的初始大小计算采用下面的公式：

```
Default Xms = MaxRAM / InitialRAMFraction
```

计算默认最小堆的大小的方法同样也适用，`InitialRAMFraction` 参数的默认值为 64。不过，这个参数也不能完全控制堆的初始值，如果该参数值小于 5 MB，或者更确切地说，指定的 `InitialRAMFraction` 小于 `-XX:OldSize=N` 参数的设定（该参数默认为 4 MB）时会采用另外的处理方式。这种情况下，堆的初始大小等于新生代和老年代大小之和。

快速小结

1. 大多数的机器上堆的初始空间和最大空间的默认值计算是比较直观的。
2. 达到堆大小的临界情况时，需要考虑的因素更多，计算也更加复杂。

6.5 小结

在前面的这两章中，我们花费了大量的时间深入介绍了垃圾收集（以及各种垃圾收集方法）工作的细节。如果垃圾收集花费的时间超出了你的预期，了解垃圾收集的内部工作原理能帮你决定采取哪些必要的步骤进行性能调优。

现在我们已经了解了所有的细节，让我们回退一步，决定选择什么方法，采用什么标志来

调优垃圾收集器。下面是一些问题集合，在调优之前，先试着回答这些问题，它们能帮你理清思路，选择合适的调优措施。

你的应用能够忍受 Full GC 的停顿吗?

如果答案是肯定的，选择 Throughput 收集器能获得最高的性能，同时，使用的 CPU 和堆的大小也比其他的垃圾收集器少。如果答案是否定的，你需要依据可用的堆大小做选择，如果可用的堆较小，你可以选择并发收集器，譬如 CMS 收集器或者 G1 收集器；如果可用的堆比较大，推荐使用 G1 收集器。

使用默认设置能达到你期望的性能目标吗?

尽量首先使用默认的设置。因为垃圾收集技术在不断发展成熟，自动调优大多数情况下取得的效果是最好的。如果使用默认设置没有达到你需要的性能目标，请确认垃圾收集是否是性能瓶颈。查看垃圾收集日志能帮我们了解 JVM 在垃圾收集上花费了多长时间、垃圾收集发生的频率是多少。对于负荷较高的应用，如果 JVM 花在垃圾收集上的时间不超过 3%，即使进行垃圾调优也不会得到太大的性能提升（不过，如果那些指标是你关注的方面，你仍然可以尝试通过调优缩短某些指标）。

应用的停顿时间与你预期的目标接近吗?

如果停顿时间与你预期的目标很接近，调整最大停顿时间的设定可能是你需要做的。如果不是，你需要进行其他的调整。如果停顿时间太长，但是应用的吞吐量正常，你可以尝试减小新生代的大小（如果瓶颈是 Full GC 的停顿，就减小老年代的大小）；调整之后，停顿的频率会增加，但是单次停顿的时长会变短。

虽然 GC 的停顿时间已经非常短了，但应用的吞吐量依旧上不去?

这种情况下你需要增大堆的大小（至少要增大新生代）。但是，这并不意味着堆越大越好：更大的堆会导致更长的停顿时间。即便是并发收集器，默认情况下，增大堆也还是意味着增大新生代，因此你会发现新生代的停顿时间变长了。即便是这样，如果有可能，还是应该增大堆的大小，或者增大对应代的大小。

你使用并发收集器吗? 是否发生了由并发模式失败引起的 Full GC ?

如果你有足够的 CPU 资源，可以尝试增加并发 GC 线程的数量，或者通过调整 `InitiatingHeapOccupancyPercent` 参数在更早的时候启动后台清理线程。对于 G1 收集器，如果有混合式垃圾收集尚未完成，并发周期就不会启动。在这个时候，可以尝试降低混合式 GC 的回收目标（Mixed GC count target）。

你使用并发收集器吗? 是否发生了由晋升失败引起的 Full GC ?

在 CMS 收集器中，发生晋升失败意味着堆发生了碎片化。这种情况下，我们能做的事情不多：使用更大的堆，或者尽早地启动后台回收都能在一定程度上缓解堆的碎片化。处理这种情况，更好的解决方法可能是使用 G1 收集器。G1 收集器中，疏散失败（To 空间溢出）表明遭遇了同样的情况，但是 G1 收集器能解决碎片化的问题，如果它的后台线程在更早的时候启动，且混合式 GC 的速度更快的话。你可以尝试通过增大并发 G1 收集线程的数目，调整 `InitiatingHeapOccupancyPercent`，或者降低混合式 GC 的目标来解决 G1 收集器中堆碎片化的问题。

第 7 章

堆内存最佳实践

第 5 章和第 6 章详细探讨了如何调优垃圾收集器，以使其尽可能不影响程序。调优垃圾收集器非常重要，但是应用更好的编程实践往往可以获得更好的性能。本章就探讨一些在 Java 中使用堆内存的最佳实践方法。

这里有两个相互冲突的目标。第一个一般规则是，有节制地创建对象并尽快丢弃。使用更少的内存，这是提升垃圾收集器效率的最好方法。相反，频繁地重建某类对象会导致整体性能变得更糟（即便 GC 的性能有所改进）。然而，如果重用那些对象，程序的性能则有可能会得到改善。对象的重用方式有很多种，包括线程局部变量、特殊的对象引用以及对象池。重用对象意味着这些对象会长期存活，而且会影响垃圾收集器，但如果能合理利用，整体性能就会得到改进。

本章会探讨这两种方法以及在它们之间的权衡。不过，我们首先要看一下可以帮助理解堆内正在发生什么的工具。

7.1　堆分析

第 5 章探讨的 GC 日志和工具对于理解 GC 对应用的影响很有帮助，但是要想获得更多信息，我们必须研究堆本身。本节探讨的工具可为我们理解应用中正在使用的对象提供帮助。

大多数情况下，这些工具仅对堆中的活跃对象有效——会在下一次 Full GC 周期内被回收的对象不会包含在工具的输出中。在某些情况下，这些工具会通过强制执行一次 Full GC 来实现其功能，所以在使用工具后，应用的行为会受到影响。而在其他一些情况下，这些工具会对堆进行走查，报告活跃对象数据，但是不会释放对象。不过不管是哪种情况，这些工具都需要一些时间和机器资源；因此一般不用于测量程序的执行。

7.1.1 堆直方图

减少内存使用是一个重要目标，但和大多数性能优化主题一样，它有助于我们把目标放在可用内存的最大化上。在本章后面，我会围绕一个 `Calendar` 对象的延迟初始化演示一个例子。这个例子会节省 640 字节的堆内存，但如果应用只初始化一个这样的对象，那么性能并不会有多大差别。我们必须通过分析来确定哪类对象消耗了大量内存。

最简单的方法是利用堆直方图。利用直方图，我们可以快速看到应用内的对象数目，同时不需要进行完整的堆转储（因为堆转储需要一段时间来分析，而且会消耗大量磁盘空间）。如果应用中的内存压力是由一些特定的对象类型引起的，利用堆直方图我们很快就能看出端倪。

堆直方图可以使用 `jcmd` 命令获得（这里使用了进程 ID 8898）：

```
% jcmd 8998 GC.class_histogram
8898:

 num     #instances       #bytes  class name
----------------------------------------------
   1:        789087     31563480  java.math.BigDecimal
   2:        237997     22617192  [C
   3:        137371     20696640  <constMethodKlass>
   4:        137371     18695208  <methodKlass>
   5:         13456     15654944  <constantPoolKlass>
   6:         13456     10331560  <instanceKlassKlass>
   7:         37059      9238848  [B
   8:         10621      8363392  <constantPoolCacheKlass>
```

在堆直方图中，`Klass` 相关的对象往往接近顶端，它们是加载类得到的元数据对象。在接近顶端的地方，字符数组（`[C`）和 `String` 对象也很常见，因为它们是最常创建的 Java 对象。字节数组（`[B`）和 `Object` 数组（`[Ljava.lang.Object;`）同样较为常见，因为类加载器会将其数据保存到这些结构中。（如果你不熟悉这里使用的语法，它来自 Java 原生接口——即 JNI——识别对象类型的方式；更多详情，请查阅 JNI 参考文档。）

在这个例子中（这里运行的是一个应用服务器中的示例股票应用的变种），其中包含的 `BigDecimal` 类就是我们要追查的东西：我们知道，该示例代码会产生大量短暂的 `BigDecimal` 对象，但是有这么多停留在堆中，这通常不是我们希望看到的情况。尽管该命令不会强制执行 Full GC，但是 `GC.class_histogram` 中的输出仅包含活跃对象。

运行下面的命令会得到类似输出：

```
% jmap -histo process_id
```

`jmap` 的输出中包含会被回收的对象（死对象）。要在看到直方图之前强制执行一次 Full GC，可以转而运行下面的命令：

```
% jmap -histo:live process_id
```

直方图非常小，所以在自动化系统中为每个测试收集一个会很有帮助。因为获得直方图也需要几秒钟，所以请不要在性能测量稳定的状态下获得。

7.1.2　堆转储

直方图擅长识别由分配了一两个特定类的过多实例所引发的问题，但是要进行深度分析，就需要堆转储了。有很多可以查看堆转储的工具，而且大多数都可以连接到运行的程序来生成转储文件。从命令行生成转储文件往往更容易，可以在下面两条命令中选择一个：

```
% jcmd process_id GC.heap_dump /path/to/heap_dump.hprof
```

或：

```
% jmap -dump:live,file=/path/to/heap_dump.hprof process_id
```

在 `jmap` 中包含 `live` 选项，这会在堆被转储之前强制执行一次 Full GC；`jcmd` 默认就会这么做。如果因为某些原因，你希望包含其他对象（即死对象），可以在 `jcmd` 命令的最后加上 `-all`。

这两条命令都会在指定目录下创建一个命名为 heap_dump.hprof 的文件。生成之后，有很多工具可以打开该文件。以下是三个最常用的工具。

`jhat`

　这是最原始的堆分析工具。它会读取堆转储文件，并运行一个小型的 HTTP 服务器，该服务器允许你通过一系列网页链接查看堆转储信息。

`jvisualvm`

　`jvisualvm` 的监视（Monitor）选项卡可以从一个运行中的程序获得堆转储文件，也可以打开之前生成的堆转储文件。在这里，我们可以浏览堆，检查最大的保留对象，以及执行任意针对堆的查询。

`mat`

　开源的 EclipseLink 内存分析器工具（EclipseLink Memory Analyzer Tool，`mat`）可以加载一个或多个堆转储文件并执行分析。它可以生成报告，向我们建议可能存在问题的地方；也可以用于浏览堆，并对堆执行类 SQL 的查询。

对堆的第一遍分析通常涉及保留内存。一个对象的保留内存，是指回收该对象可以释放出的内存量。在图 7-1 中，String Trio 对象的保留内存包括该对象本身占用的内存，以及 Sally 和 David 两个对象占用的内存。Michael 对象占用的内存不在此列，如果 String Trio 被释放，因为 Michael 对象还有另一个指向它的引用，所以并不满足 GC 条件。

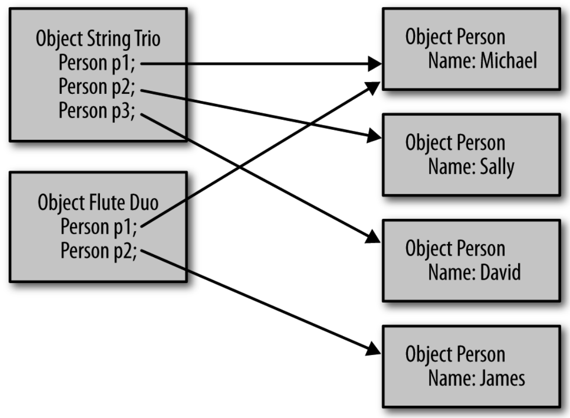


图 7-1：保留内存的对象图

浅对象大小、保留对象大小及深对象大小

对于内存分析，还有其他两个很有用的术语：浅（shallow）和深（deep）。一个对象的浅大小，指的是该对象本身的大小。如果该对象包含一个指向另一个对象的引用，4 字节或 8 字节的引用会计算在内，但是目标对象的大小不会包含进来。

深大小则包含那些对象的大小。深大小与保留大小的区别在于那些存在共享的对象。在图 7-1 中，Flute Duo 对象的深大小包括 Michael 对象消耗的内存空间，但是保留大小则不包括。

保留了大量堆空间的对象一般称作堆的支配者。如果堆分析工具表明有些对象支配着大部分堆，那事情就好办了：我们需要做的就是少创建一些这类对象，减少保留这类对象的时间，简化其对象图，或者将对象变小。可能说起来容易做起来难，但是至少分析起来很简单。

更普遍的情况是，因为程序可能会共享对象，所以有时必须做一些侦查性工作。就像图 7-1 中的 Michael 对象一样，那些共享的对象不会计算在其他任何对象的保留集内，因为单独释放一个对象并不会释放共享对象。此外，最大的保留大小往往是我们几乎无法控制的类加载器带来的。举一个极端的例子，我们以运行在 GlassFish 中的某个版本的股票小服务程序为例，有些条目以强引用形式缓存在用户会话中，同时以弱引用形式保存在一个全局的散列表中（所以缓存的条目存在多个指向它们的引用），图 7-2 显示了堆中位列前茅的一些保留对象。

Class Name	Shallow Heap	Retained Heap	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
org.apache.felix.bundlerepository.impl.LocalRepositoryImpl @ 0x77	32	6,537,744	0.43%
org.apache.felix.framework.BundleWiringImpl$BundleClassLoader	96	5,446,344	0.36%
org.jvnet.hk2.osgiadapter.OSGiModulesRegistryImpl @ 0x77d3fc6a0	64	4,894,168	0.32%
com.sun.tools.javac.file.ZipFileIndex @ 0x7827d5fa0	88	2,384,344	0.16%
org.apache.felix.framework.BundleWiringImpl$BundleClassLoader	96	1,453,056	0.10%
net.sdo.stockimpl.StockPriceHistoryImpl @ 0x7c018c868	48	1,357,544	0.09%
com.sun.tools.javac.file.ZipFileIndex @ 0x78301f4c0	88	1,346,072	0.09%
net.sdo.stockimpl.StockPriceHistoryImpl @ 0x7a27a59a0	48	1,334,664	0.09%
org.apache.felix.framework.BundleWiringImpl$BundleClassLoader	96	1,331,296	0.09%
net.sdo.stockimpl.StockPriceHistoryImpl @ 0x788769d38	48	1,328,368	0.09%
net.sdo.stockimpl.StockPriceHistoryImpl @ 0x7acfd9098	48	1,327,776	0.09%
net.sdo.stockimpl.StockPriceHistoryImpl @ 0x79d051d88	48	1,322,528	0.09%
net.sdo.stockimpl.StockPriceHistoryImpl @ 0x7a71fe2b8	48	1,321,344	0.09%
net.sdo.stockimpl.StockPriceHistoryImpl @ 0x7c32cada0	48	1,319,480	0.09%
Σ Total: 14 of 70,584 entries; 70,570 more			

图 7-2：在 Memory Analyzer 中查看保留内存

堆中大约有 1.4 GB 的对象（这个值没有出现在该选项卡上）。即便如此，单向引用的最大的一组对象只有 6 MB（而且不出所料，这是 GlassFish 的 OSGi 类加载框架的一部分）。看了直接保留内存最多的一些对象，这对解决内存问题并没有什么帮助。

在这个例子中，列表中有多个 `StockPriceHistoryImpl` 实例，而且每一个都保留了相当数量的内存。从这些对象消耗的内存量可以推断出，它们就是问题所在。不过在一般情况下，对象可能会以这样的方式被共享，所以从保留堆看不出任何很明显的东西。

直方图用在第二步很有用（参见图 7-3）。

Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
java.math.BigDecimal	12,920,067	516,802,680	517,429,776
java.util.TreeMap$Entry	7,255,390	290,215,600	1,450,796,576
net.sdo.stockimpl.StockPriceImpl	7,240,530	289,621,200	980,225,584
java.util.Date	7,244,268	173,862,432	174,077,552
net.sdo.stockimpl.StockPricePK	7,240,530	173,772,720	173,799,360
char[]	266,992	25,934,280	25,934,280
java.lang.String	255,336	6,128,064	30,780,696
java.util.HashMap$Entry[]	59,102	5,050,328	30,515,800
java.util.HashMap$Entry	151,237	4,839,584	30,295,176
java.util.LinkedHashMap$Entry	72,786	2,911,440	6,298,496
com.sun.tools.javac.file.ZipFileIndex$Entry	44,416	2,131,968	6,049,552
java.lang.Object[]	31,328	1,930,928	23,857,992
java.util.HashMap	34,114	1,910,384	29,772,824
java.lang.reflect.Method	21,579	1,726,320	3,714,040
Σ Total: 14 of 12,007 entries; 11,993 more	**43,446,283**	**1,517,322,152**	

图 7-3：在 Memory Analyzer 中查看直方图

直方图将同一类型的对象聚合到了一起，而且在这个例子中更容易看出来，这 1.4 GB 的内存被 700 万个 `TreeMap$Entry` 对象占据着，而这正是关键所在。即便不知道程序内部目前的运转情况，使用 Memory Analyzer 的工具来跟踪这些对象，看看它们保持了哪些东西，这也相当直观了。

堆分析工具为找到某个特定对象（或者这个例子中的一组对象）的 GC 根提供了一种方法——尽管直接跳到 GC 根未必有多大帮助。GC 根是一些系统对象，其中保存着一些（通过一个较长的由其他对象组成的链条）指向问题中对象的静态和全局引用。它们通常来自在系统或 bootstrap 类路径下加载的某个类中的静态变量，其中包括 `Thread` 类和所有的活跃线程；线程保留对象，或是通过其线程局部变量，或是通过目标 `Runnable` 对象（或者在存在 `Thread` 的子类的情况下，通过子类中包含的任何其他引用）来引用。

在某些情况下，知道某个目标对象的 GC 根是有用的，但是如果有多个指向该对象的引用，那么它会有多个 GC 根。这里的引用是一个倒过来的树结构。假设有两个对象指向一个特定的 `TreeMap$Entry` 对象。其中每个对象又可能被其他两个对象引用，而其他两个对象引用中的每一个，还有可能被另外三个对象引用，诸如此类。引用会随着追根溯源的过程爆炸性增长，这意味着任何给定的对象都可能有多个 GC 根。

相反，在对象图中，检查并找出对象被共享的最下面的一点可能更富成效。实现方法是检查对象及指向该对象的引用，然后跟踪这些引用，直到识别出重复的路径。在这个例子中，两个地方用到了保存在 Tree Map 中的 `StockPriceHistoryImpl` 对象：一个是 `ConcurrentHashMap`，它保存着会话数据；一个是 `WeakHashMap`，它保存着全局缓存。

在图 7-4 中，展开追溯就足以显示这两个类的一点数据了。要得出它是会话数据的结论，我们的方法是继续展开 `ConcurrentHashMap` 的路径，直到可以明显看出该路径是会话数据这一点。`WeakHashMap` 的路径也使用了类似逻辑。

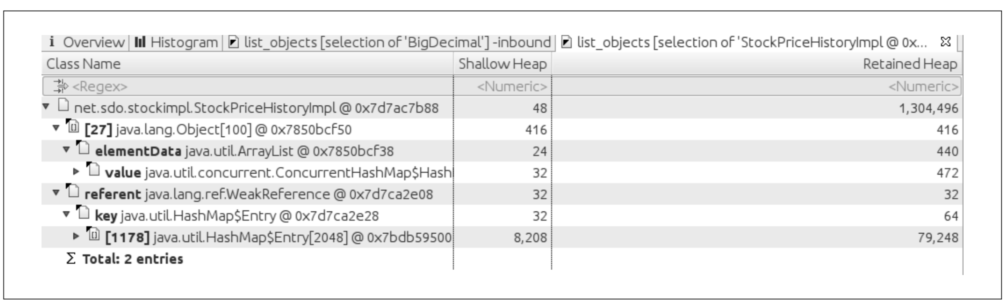


图 7-4：在 Memory Analyzer 中追溯对象引用

这个例子中用到的对象类型使分析要比通常情况下更容易一些。如果这个应用中的主要数据被建模为 `String` 对象，而非 `BigDecimal` 对象，而且保存在 `HashMap` 对象而非 `TreeMap` 对象中，分析会更困难。因为堆转储中会有数十万其他字符串，成千上万的其他 `HashMap` 对象，找到通往我们关注的那些对象的路，着实需要一些耐心。一般来说，我们要从集合类对象（例如 `HashMap`）入手，而不是从记录项（例如 `HashMap$Entry`）入手，并且要寻找最大的集合。

快速小结

1. 了解哪些对象正在消耗内存，是了解要优化代码中哪些对象的第一步。
2. 对于识别由创建了太多某一特定类型对象所引发的内存问题，直方图这一方法快速且方便。
3. 堆转储分析是追踪内存使用的最强大的技术，不过要利用好，则需要一些耐心和努力。

7.1.3　内存溢出错误

在下列情况下，JVM 会抛出内存溢出错误（`OutOfMemoryError`）：

- JVM 没有原生内存可用；
- 永久代（在 Java 7 和更早的版本中）或元空间（在 Java 8 中）内存不足；
- Java 堆本身内存不足——对于给定的堆空间而言，应用中活跃对象太多；
- JVM 执行 GC 耗时太多。

后面两种情况涉及 Java 堆本身，它们更为常见，但是不要看到 `OutOfMemoryError` 就自动下结论认为堆是问题所在。你必须看一下为什么会发生这种错误（原因会是异常输出的一部分）。

1. 原生内存不足

列表中的第一种情况——JVM 没有原生内存可用，其原因与堆根本无关。在 32 位的 JVM 中，一个进程的最大内存是 4 GB（在某些版本的 Windows 上是 3 GB，在某些比较老的 Linux 版本上是 3.5 GB）。指定一个非常大的堆大小，比如说 3.8 GB，使应用的大小很接近 4 GB 的限制，这很危险。即便在 64 位的 JVM 中，操作系统的虚拟内存也不是 JVM 请求多少就有多少。

第 8 章会更完整地介绍这个主题。你必须意识到，如果 `OutOfMemoryError` 消息中提到了原生内存的分配，那对堆进行调优解决不了问题：你需要看一下错误中提到的是何种原生内存问题。例如，下面的消息说明线程栈的原生内存耗尽了：

```
Exception in thread "main" java.lang.OutOfMemoryError:
unable to create new native thread
```

2. 永久代或元空间内存不足

这种内存错误与堆无关，其发生原因是永久代（在 Java 7 中）或元空间原生内存（在 Java 8 中）满了。根源可能有两种情况：第一种情况是应用使用的类太多，超出了永久代的默认容纳范围；解决方案是增加永久代的大小（参见 5.2.3 节）。（在 Java 8 中，如果设置了类的元空间的最大大小，也会出现同样的问题。）

第二种情况相对来说有点棘手：它涉及类加载器的内存泄漏。这种情况经常出现于 Java EE 应用服务器中。部署到应用服务器中的每个应用都运行在自己的类加载器中（这提供了隔离，使一个应用中的类不会和另一个应用中的类共享，也不会有干扰）。在开发中，每次修改了应用都必须重新部署，这时就会创建一个新的类加载器来加载新的类，而老的类加载器就可以退出作用域了。一旦类加载器退出了作用域，该类的元数据就可以回收了。

如果老的类加载器没有退出作用域，那么该类的元数据也就无法释放，最后永久代就会被填满，进而抛出 OutOfMemoryError。在这种情况下，增加永久代的大小会有所帮助，但最终只是推迟了错误抛出的时机而已。

如果在某个应用服务器环境中出现这种情况，除了联系应用服务器厂商，让他们修复内存泄漏问题外，也别无他法。如果你正在编写的应用会创建并丢弃大量类加载器，一定要非常谨慎，确保类加载器本身能正确丢弃（尤其是，确保没有线程将其上下文加载器设置成一个临时的类加载器）。要调试这种情况，前面刚介绍的堆转储分析就非常有用：在直方图中，找到 ClassLoader 类的所有实例，然后跟踪它们的 GC 根，看一下哪些对象还保留了对它们的引用。

识别这种情况的关键仍然是 OutOfMemoryError 的输出全文。在 Java 8 中，如果元空间满了，错误消息将会是下面这样的：

```
Exception in thread "main" java.lang.OutOfMemoryError: Metaspace
```

在 Java 7 中类似，错误消息如下：

```
Exception in thread "main" java.lang.OutOfMemoryError: PermGen space
```

3. 堆内存不足

当确实是堆本身内存不足时，错误消息会是这样的：

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```

由缺乏堆空间触发的内存不足条件一般与永久代的情况类似。应用可能只是需要更多堆空间：活跃对象的数目在为其配置的堆空间中已经装不下了。也可能是应用存在内存泄漏：它持续分配新对象，却没有让其他对象退出作用域。对于第一种情况，增加堆大小可以解决问题；而对于第二种情况，增加堆大小只不过将错误的出现时机推迟了。

不管是哪种情况，要找出哪些对象消耗的内存最多，堆转储分析都是必要的；之后我们的注意力就可以集中到减少那些对象的数目（或大小）上。如果应用存在内存泄漏，可以间隔几分钟，获得连续的一些堆转储文件，然后加以比较。mat 内置了这一功能：如果打开了两个堆转储文件，mat 有一个选项用来计算两个堆中的直方图之间的差别。

自动堆转储

OutOfMemoryError 是不可预料的，我们很难确定应该何时获得堆转储。有几个 JVM 标志可以起到帮助。

-XX:+HeapDumpOnOutOfMemoryError

该标志默认为 false，打开该标志，JVM 会在抛出 OutOfMemoryError 时创建堆转储。

-XX:HeapDumpPath=<path>

该标志指定了堆转储将被写入的位置；默认会在应用的当前工作目录下生成 java_pid<pid>.hprof 文件。这里的路径可以指定目录（这种情况下会使用默认的文件名），也可以指定要生成的实际文件的名字。

`-XX:+HeapDumpAfterFullGC`

这会在运行一次 Full GC 后生成一个堆转储文件。

`-XX:+HeapDumpBeforeFullGC`

这会在运行一次 Full GC 之前生成一个堆转储文件。

有的情况下（比如，因为执行了多次 Full GC）会生成多个堆转储文件，这时 JVM 会在堆转储文件的名字上附一个序号。

如果应用会因为堆空间的原因不可预测地抛出 `OutOfMemoryError`，而且你需要那一刻的堆转储来分析错误原因，请尝试打开这些标志。

图 7-5 演示了由集合类（这里是一个 `HashMap`）引发的 Java 内存泄漏这一经典案例。（集合类是导致内存泄漏的最常见原因：应用向集合中插入条目，但从不释放它们。）这是一个直方图对比视图：它显示了两个不同的堆转储中对象数目的差别。例如，与基线堆转储相比，目标堆转储中的 `Integer` 对象要多出 19 744 个。

要克服这种情况，最好的办法是修改应用的逻辑，主动将不再需要的条目从集合中删除。作为一种选择，可以使用弱引用或软引用的集合，当在应用中已经不存在对某些条目的任何引用时，该集合会自动丢弃它们，不过这样的集合是有代价的（本章后面会讨论）。

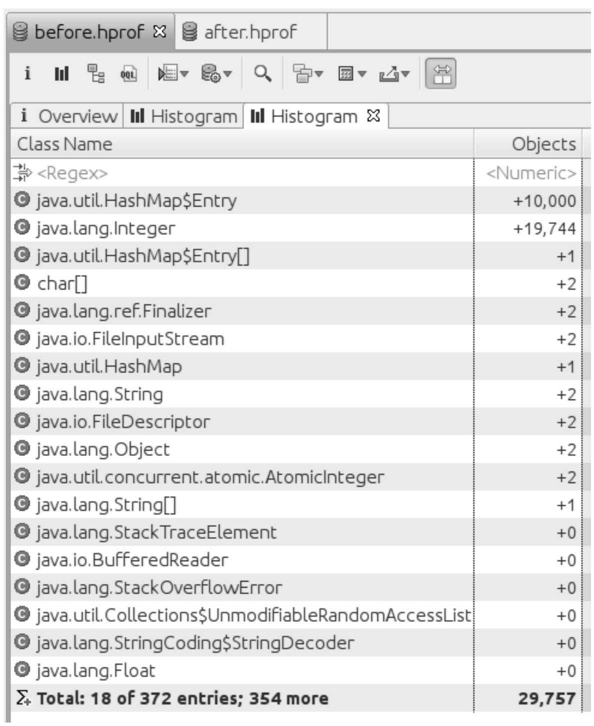


图 7-5：直方图对比

4. 达到GC的开销限制

JVM 抛出 `OutOfMemoryError` 的最后一种情况是 JVM 认为在执行 GC 上花费了太多时间：

```
Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded
```

当满足下列所有条件时就会抛出该错误。

(1) 花在 Full GC 上的时间超出了 `-XX:GCTimeLimit=N` 标志指定的值。其默认值是 98（也就是，如果 98% 的时间花在了 GC 上，则该条件满足）。
(2) 一次 Full GC 回收的内存量少于 `-XX:GCHeapFreeLimit=N` 标志指定的值。其默认值是 2，这意味着如果 Full GC 期间释放的内存不足堆的 2%，则该条件满足。
(3) 上面两个条件连续 5 次 Full GC 都成立（这个数值是无法调整的）。
(4) `-XX:+UseGCOverhead-Limit` 标志的值为 `true`（默认如此）。

请注意，所有四个条件必须都满足。一般来说，应用中连续执行了 5 次以上的 Full GC，不一定会抛出 `OutOfMemoryError`。其原因是，即便应用将 98% 的时间花费在执行 Full GC 上，但是每次 GC 期间释放的堆空间可能会超过 2%。这种情况下可以考虑增加 `GCHeapFreeLimit` 的值。

还请注意，如果前两个条件连续 4 次 Full GC 周期都成立，作为释放内存的最后一搏，JVM 中所有的软引用都会在第五次 Full GC 之前被释放。这往往会防止该错误，因为第五次 Full GC 很可能会释放超过 2% 的堆内存（假设该应用使用了软引用）。

快速小结

1. 有多种原因会导致抛出 `OutOfMemoryError`，因此不要假设堆空间就是问题所在。
2. 对于永久代和普通的堆，内存泄漏是出现 `OutOfMemoryError` 的最常见原因；堆分析工具可以帮助我们找到泄漏的根源。

7.2 减少内存使用

在 Java 中，第一种更高效使用内存的方式是减少堆内存的使用。这句话不难理解：堆内存用的越少，堆被填满的几率就越低，需要的 GC 周期也越少。而且有倍乘效应：新生代回收的次数更少，对象的晋升年龄也就不会很频繁地增加，这意味着对象被提升到老年代的可能性也降低了。因此，Full GC 周期（或者是并发 GC 周期）也会减少。而且，如果这些 Full GC 周期能够清理更多内存，它们发生的频率也会降低。

本节将研究三种减少内存使用的方式：减少对象大小、对象的延迟初始化以及使用规范化对象。

7.2.1 减少对象大小

对象会占用一定数量的堆内存，所以要减少内存使用，最简单的方式就是让对象小一些。考虑运行程序的机器的内存限制，增加 10% 的堆有可能是无法做到的，但是堆中一半对象的大小减少 20%，能够实现同样的目标。

减少对象大小有两种方式：减少实例变量的个数（效果很明显），或者减少实例变量的大小（效果没那么明星）。表 7-1 列出了 Java 中不同类型实例变量的大小。

表7-1：Java实例变量的大小

类型	大小
byte	1
char	2
short	2
int	4
float	4
long	8
double	8
reference	在 32 位 JVM 以及堆小于 32 GB 的 64 位 JVM 上是 4；在启用大堆的 64 位 JVM 上是 8[a]

注 a：更多细节，参见 8.2.2 节“压缩的 oop”。

这里的引用类型指的是指向任何类型 Java 对象（包括类或数组的实例）的引用。这个空间存储的只是参数本身。如果对象中包含指向其他对象的引用，其大小会因我们想考虑 Shallow Size、Deep Size 还是 Retained size（保留大小）而有所不同，不过其中都会包含一些隐藏的对象头字段。对于普通对象，对象头字段在 32 位 JVM 上占 8 字节，在 64 位 JVM 上占 16 字节（跟堆大小无关）。对于数组，对象头字段在 32 位 JVM 以及堆小于 32 GB 的 64 位 JVM 上占 16 字节，其他情况下是 64 字节。

例如，考虑这几个类定义：

```
public class A {
    private int i;
}

public class B {
    private int i;
    private Locale l = Locale.US;
}

public class C {
    private int i;
    private ConcurrentHashMap chm = new ConcurrentHashMap();
}
```

在堆小于 32 GB 的 64 位 Java 7 JVM 上，这几个类的实例实际大小如表 7-2 所示。

表7-2：简单对象的大小

	Shallow size	Deep size	Retained size
A	16	16	16
B	24	216	24
C	24	200	200

在 B 类中，定义 Locale 应用将对象的大小增加了 8 字节，但至少在这个例子中，实际的 Locale 对象是与其他一些类共享的。如果该类实际上从来没用到这个 Locale 对象，那将这个实例包含进来，只会浪费引用所占的额外空间。当然，如果应用创建了大量 B 类的实例，还是会积少成多。

另一方面，定义并创建一个 ConcurrentHashMap，除了对象应用会消耗额外的字节，这个 HashMap 对象还会增加 200 字节。如果这个 HashMap 从来不用，C 的实例就非常浪费。

仅定义需要的实例变量，这是节省对象空间的一种方式。还有一种效果不那么明显的方案，就是使用更小的数据类型。如果某个类需要记录 8 个可能的状态之一，用一个字节就可以了，而不需要一个 int，这就可能会节省 3 字节。使用 float 代替 double，int 代替 long，诸如此类，都可以帮助节省内存，特别是在那些会频繁地实例化的类中。第 12 章将讨论，使用大小适当的集合类（或者使用简单的实例变量代替集合类）可以达到类似的节省空间的目的。

对象对齐与对象大小

表 7-2 中的类，都包含一个额外的整型字段，讨论中并没有引用到。为什么要放这么一个变量呢？

事实上，这个变量的目的是让讨论更容易理解：B 类比 A 类多 8 字节，正是我们所期望的（这样更明确）。

这掩盖了一个重要细节：为使对象大小是 8 字节的整数倍（对齐），总是会有填充操作。如果在 A 类中没有定义 i，A 的实例仍然会消耗 16 字节，其中 4 字节只是用于填充，使得对象大小是 8 的整数倍，而不是用于保存 i。如果没有定义 i，B 类的实例将仅消耗 16 字节，和 A 一样，即便 B 中还有额外的对象引用。B 中仅包含一个额外的 4 字节引用，为什么其实例会比 A 的实例多 8 字节呢，也是填充的问题。

JVM 也会填充字节数不规则的对象，这样不管底层架构最适合什么样的地址边界，对象的数组都能优雅地适应。

因此，去掉某个实例字段或者减少某个字段的大小，未必能带来好处，不过我们没有理由不这么做。

去掉对象中的实例字段，有助于减少对象的大小，不过还有一个灰色地带：有些字段会保存基于一些数据计算而来的结果，这该如何处理呢？这就是计算机科学中典型的时间空间权衡问题：是消耗内存（空间）保存这个值更好，还是在需要时花时间（CPU 周期）计算这个值更好？不过在 Java 中，权衡还会考虑 CPU 时间，因为额外的内存占用会引发 GC 消耗更多 CPU 周期。

比如，String 的哈希码值（hashcode）就是对一个涉及该字符串中每个字符的式子求和计算而来的；计算会消耗一点时间。因此，String 类会把这个值存在一个实例变量中，这样哈希码值只需要计算一次：最后，与不存储这个值而节省的内存空间相比，重用几乎总能获得更好的性能。另一方面，大部分类的 toString() 方法不会把对象的字符串表示保存在一个实例变量中，因为实例变量及其引用的字符串都会消耗内存。相反，与保存字符串引

用所需的内存相比，计算一个新的字符串所花的时间通常不是很多，性能更好。（还有一个因素，String 对象的哈希码值用的较为频繁，而对象的 toString() 表示使用却很少。）

当然，这种情况必定是因人而异的。就时间 / 空间的连续体而言，究竟是使用内存来存储值，还是重新计算值，都是取决于许多具体因素的。如果目标是减少 GC，则更倾向于采用重新计算。

快速小结

1. 减小对象大小往往可以改进 GC 效率。
2. 对象大小未必总能很明显地看出来：对象会被填充到 8 字节的边界，对象引用的大小在 32 位和 64 位 JVM 上也有所不同。
3. 对象内部即使为 null 的实例变量也会占用空间。

7.2.2 延迟初始化

正如前面几节所介绍的，很多时候，决定一个特定的实例变量是否需要并不是非黑即白的问题。某个特定的类可能只有 10% 时间需要一个 Calendar 对象，但是 Calendar 对象创建成本很高，所以保留这个对象备用，而不是需要的时候再重新创建，绝对是有意义的。这种情况下，延迟初始化可以带来帮助。

到目前为止，我们所作讨论的前提是假定实例变量很早就会初始化。需要使用一个 Calendar 对象（不需要线程安全）的类看上去可能是这样的：

```
public class CalDateInitialization {
    private Calendar calendar = Calendar.getInstance();
    private DateFormat df = DateFormat.getDateInstance();

    private void report(Writer w) {
        w.write("On " + df.format(calendar.getTime()) +": " + this);
    }
}
```

要延迟初始化其字段，在计算性能上会有一点小小的损失，代码每次执行时都必须测试变量的状态：

```
public class CalDateInitialization {
    private Calendar calendar;
    private DateFormat df;

    private void report(Writer w) {
        if (calendar == null) {
            calendar = Calendar.getInstance();
            df = DateFormat.getDateInstance();
        }
        w.write("On " + df.format(calendar.getTime()) +": " + this);
    }
}
```

如果问题中的这个操作使用不太频繁，那延迟初始化最适合：如果操作很常用，实际上没有节省内存（总是会分配这些实例），而常用操作又有轻微的性能损失。

延迟初始化运行时性能

检查要进行延迟初始化的变量是不是已经被初始化了，未必总会有性能损失。考虑来自 JDK 的 ArrayList 类的一个例子。这个类维护着一个所存储元素的数组，在 JDK 7u40 之前，这个类的伪代码看上去就是下面这样：

```
public class ArrayList {
    private Object[] elementData = new Object[16];
    int index = 0;
    public void add(Object o) {
        ensureCapacity();
        elementData[index++] = o;
    }
    private void ensureCapacity() {
        if (index == elementData.length) {
            ……重新分配数组并把老数据复制进来……
        }
    }
}
```

在 JDK 7u40 中，这个类有所修改，elementData 数组被延迟初始化了。但是因为 ensureCapacity() 方法已经需要检查数组大小，这个类的常用方法就不用承受性能损失了：检查是否初始化的代码和检查数组大小是否需要增加的代码是一样的。新的代码使用了一个静态的、共享的 0 长度数组，因此性能也是一样的：

```
public class ArrayList {
    private static final Object[] EMPTY_ELEMENTDATA = {} ;
    private Object[] elementData = EMPTY_ELEMENTDATA;
}
```

这意味着 ensureCapacity() 方法基本不需要修改，因为 index 和 elementData.length 都是从 0 开始的。

当所涉及的代码需要保证线程安全时，延迟初始化会更为复杂。第一步，最简单的方式是添加传统的同步机制：

```
public class CalDateInitialization {
    private Calendar calendar;
    private DateFormat df;

    private synchronized void report(Writer w) {
        if (calendar == null) {
            calendar = Calendar.getInstance();
            df = DateFormat.getDateInstance();
        }
        w.write("On " + df.format(calendar.getTime()) +": " + this);
    }
}
```

在解决方案中引入同步，会使得同步也有可能成为性能瓶颈。不过这种情况很罕见。对于问题中的对象而言，只有当初始化这些字段的几率很低时，延迟初始化才有性能方面的好

处。因为，如果一般情况下都会初始化这些字段，那实际上也不会节省内存。因此对于延迟初始化的字段，当不常用的代码路径突然被大量线程同时使用时，同步就会成为瓶颈。这种情况是可以想象的，不过好在并不多见。

只有延迟初始化的变量本身是线程安全的，才有可能解决同步瓶颈。DateFormat 对象不是线程安全的，所以在现在的这个例子中，锁中是否包含 Calendar 对象并不重要：如果延迟初始化的对象突然被频频使用，那无论如何，围绕 DateFormat 对象所需的同步都会成为问题。线程安全的代码应该是这样的：

```
public class CalDateInitialization {
    private Calendar calendar;
    private DateFormat df;

    private void report(Writer w) {
        unsychronizedCalendarInit();
        synchronized(df) {
            w.write("On " + df.format(calendar.getTime()) +": " + this);
        }
    }
}
```

涉及非线程安全的实例变量的延迟初始化，总会围绕这个变量做同步（例如，像前面所示的那样使用方法的同步版本）。

考虑一个有点不一样的例子，其中有一个比较大的 ConcurrentHashMap 对象，就采用了延迟初始化：

```
public class CHMInitialization {
    private ConcurrentHashMap chm;

    public void doOperation() {
        synchronized(this) {
            if (chm == null) {
                chm = new ConcurrentHashMap();
                …… 填充这个map的代码 ……
            }
        }
        ……使用chm……
    }
}
```

因为多个线程可以安全地访问 ConcurrentHashMap，所以这个例子中的多余的同步，就是一种不太常见的情况，因为即便是恰当地使用延迟初始化，也引入了同步瓶颈。（不过这种瓶颈应该极为少见；如果这个 HashMap 访问非常频繁，那就应该考虑延迟初始化到底有什么好处了。）该瓶颈可以使用双重检查锁这种惯用法来解决：

```
public class CHMInitialization {
    private volatile ConcurrentHashMap instanceChm;

    public void doOperation() {
        ConcurrentHashMap chm = instanceChm;
        if (chm == null) {
```

```
                synchronized(this) {
                    chm = instanceChm;
                    if (chm == null) {
                        chm = new ConcurrentHashMap();
                        …… 填充这个map的代码
                        instanceChm = chm;
                    }
                }
                ……使用chm……
            }
        }
    }
```

这里有些比较重要的多线程相关的问题：实例变量必须用 `volatile` 来声明，而且将这个实例变量赋值给一个局部变量，性能会有些许改进。第 9 章会介绍更多细节；在多线程代码的延迟初始化确实有意义的特殊场合，应该遵循这种设计模式。

尽早清理

从延迟初始化变量可以推出另一种行为，即通过将变量的值设置为 `null`，实现尽早清理，从而使问题中的对象可以更快地被垃圾收集器回收。不过这只是理论上听着不错，真正能发挥作用的场合很有限。

可以选择延迟初始化的变量，可能看上去也可以选择尽早清理：在上面的例子中，一完成 `report()` 方法，`Calendar` 和 `DateFormat` 对象就可以设置为 `null` 了。然而，如果后面再调用到这个方法（或者同一个类中的其他地方）时，并没有用到该变量，那最初就没有理由将其设计为实例变量：在方法中创建一个局部变量就可以了，而且当方法完成时，局部变量就会离开作用域，然后垃圾收集器就可以释放它了。

不需要尽早清理变量，这个规则有个很常见的例外情况，即对于类似 Java 集合类框架中的那些类：它们会在较长的时间内保存一些指向数据的引用，当问题中的数据不再需要时会通知它们。考虑 JDK 中 `ArrayList` 类的 `remove()` 方法的实现（部分代码有所简化）：

```
public E remove(int index) {
    E oldValue = elementData(index);
    int numMoved = size - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, index+1,
                         elementData, index, numMoved);
    elementData[--size] = null; // 清理,让GC完成其工作
    return oldValue;
}
```

JDK 源代码中有一行关于 GC 的注释：像这样将某个变量的值设置为 `null`，这种操作并不常见，需要解释一下。在这种情况下，我们可以看看当数组的最后一个元素被移除时，会发生什么。仍然存在于数组中的条目数，也就是实例变量 `size`，会被减 1。比如说 `size` 从 5 减少到 4。现在不管 `elementData[4]` 中存的是什么，都不能访问了：它超出了数组的有效范围。

在这种情况下，`elementData[4]` 是一个过时的引用。`elementData` 数组可能仍会存活很长时间，因此对于不需要再引用的元素，应该主动将其设置为 `null`。

过时引用的概念是这里的关键：如果一个长期存活的类会缓存以及丢弃对象引用，那一定要仔细处理，以避免过时引用。否则，显式地将一个对象引用设置为 `null` 在性能方面基本没什么好处。

快速小结

1. 只有当常用的代码路径不会初始化某个变量时，才去考虑延迟初始化该变量。
2. 一般不会在线程安全的代码上引入延迟初始化，否则会加重现有的同步成本。
3. 对于使用了线程安全对象的代码，如果要采用延迟初始化，应该使用双重检查锁。

7.2.3 不可变对象和标准化对象

在 Java 中，很多对象类型都是不可变的。这包括那些有相应的基本类型的类，如 `Integer`、`Double` 和 `Boolean` 等，以及其他一些基于数值的类型，如 `BigDecimal`。当然，最常见的 Java 对象当属不可变的 `String`。从程序设计的角度看，用定制类来表示不可变的对象，往往是个不错的主意。

如果这些对象会快速创建然后丢弃，它们会对 Young GC 多少有些影响；不过如我们在第 5 章所介绍，影响有限。但是和任何对象一样，如果有大量的不可变对象被提升到老年代，性能就会出现问题。

因此，没有理由不设计和使用不可变对象，即使对象无法改变、必须重新创建等特性使其看上去有点事与愿违。不过处理这些对象时往往可以进行一项优化，那就是避免创建同一对象的不同冗余副本。

最好的例子就是 `Boolean` 类。在任何 Java 应用中，其实只需要两个 `Boolean` 示例，一个表示 `true`，一个表示 `false`。遗憾的是，`Boolean` 设计得很差。因为它有一个 `public` 的构造器，应用喜欢创建多少这类对象就能创建多少，即时它们和两个标准化的 `Boolean` 对象其中之一是完全相同的。更好的设计方案应该是，让 `Boolean` 类只有一个 `private` 的构造器，通过 `static` 方法根据其参数返回 `Boolean.TRUE` 或 `Boolean.FALSE`。如果自己的不可变类有这样的一个模型可以遵循，就可以防止它们占用应用中额外的堆空间。（很明显，绝对不应该创建 `Boolean` 对象；必要的时候应该使用 `Boolean.TRUE` 或 `Boolean.FALSE`。）

像这类不可变对象的单一化表示，就被称为对象的标准化（canonical）版本。

创建标准化对象

即便某个特定类的全体对象几乎是无限制的，使用标准化的值通常也可以节省内存。JDK 为大部分常见的不可变对象提供了实现此功能的方法：比如字符串可以调用 `intern()` 方法找到该字符串的一个标准化版本。下一节将介绍字符串保留（intern）的更多细节，现在我们先看一下对于定制的类如何实现同样功能。

要标准化某个对象，创建一个 Map 来保存该对象的标准化版本。为防止内存泄漏，务必保证使用弱引用处理 Map 中的对象。这样一个类的骨架看上去会是这样的：

```
public class ImmutableObject {
    WeakHashMap<ImmutableObject, ImmutableObject> map = new WeakHashMap();

    public ImmutableObject canonicalVersion(ImmutableObject io) {
        synchronized(map) {
            ImmutableObject canonicalVersion = map.get(io);
            if (canonicalVersion == null) {
                map.put(io, io);
                canonicalVersion = io;
            }
            return canonicalVersion;
        }
    }
}
```

在多线程环境中，此处的同步可能会成为瓶颈。如果想坚持使用 JDK 的类，并没有简单的解决方案，因为 JDK 没有提供支持弱引用的并发 Hashmap。不过，目前有提议向 JDK 中添加一个 `CustomConcurrentHashMap` 类（最初是 JSR 166 的一部分），另外还可以找一下这种类的各种第三方实现。

快速小结

1. 不可变对象为标准化（canonicalization）这种特殊的生命周期管理提供了可能性。
2. 通过标准化去掉不可变对象的冗余副本，可以极大减少应用消耗的堆内存。

7.2.4　字符串的保留

字符串无疑是最常见的 Java 对象；应用的堆中几乎到处都是字符串。

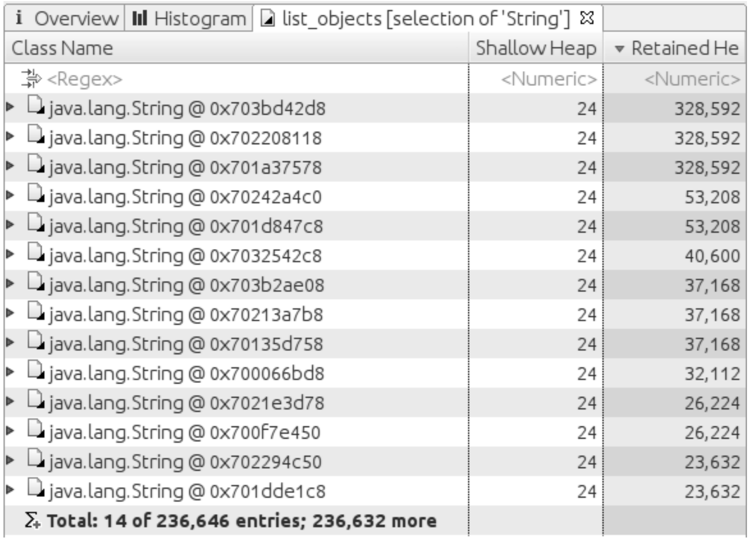
如果有大量的字符串是相同的，那很大一部分空间都是浪费的。因为字符串是不可变的，所以对于同样的字符序列，没有理由存在多个字符串表示。不过就编程而言，很难确定是不是正在创建重复的字符串。

要知道是不是有大量重复的字符串，需要对堆进行一些分析。方式之一就是在 Eclipse Memory Analyzer 中加载堆转储文件，计算所有 `String` 对象的保留大小（Retained Size），并按照其最大保留大小将这些对象排序。图 7-6 就是一个这样的堆转储信息。看上去前 3 个字符串是相同的，保留它们能够节省 650 KB 内存。（可以在验证工具中检查这些字符串。）第 4 个和第 5 个，第 7 个到第 9 个，也是这样，当然也有差别，那就是列表中越小的对象，通过保留字符串能节省的内存越少。

这种情况下，保留特定的字符串有优势；仅保留一个标准化版本，可以节省掉副本对象所消耗的空间。这可以用上一节例子中标准化例子的一个变种来实现，不过 `String` 类提供了自己的标准化方法：`intern()` 方法。

和大部分优化一样，保留字符串不能随意进行；但是如果有大量重复的字符串，占据了很大一部分堆，这时就很有效果了。关于保留太多字符串，应该注意一点：保留字符串的表是保存在原生内存中的，它是一个大小固定的 Hashtable。在 Java 7u40 之前的版本中，这

个表默认有 1009 个桶；平均而言，在因为链接而出现冲突之前，预计可以保存 500 个字符串。在 64 位版本的 Java 7u40 及更新的版本中，默认大小为 60 013。



i Overview | Histogram | list_objects [selection of 'String']

Class Name	Shallow Heap	Retained He
<Regex>	<Numeric>	<Numeric>
java.lang.String @ 0x703bd42d8	24	328,592
java.lang.String @ 0x702208118	24	328,592
java.lang.String @ 0x701a37578	24	328,592
java.lang.String @ 0x70242a4c0	24	53,208
java.lang.String @ 0x701d847c8	24	53,208
java.lang.String @ 0x7032542c8	24	40,600
java.lang.String @ 0x703b2ae08	24	37,168
java.lang.String @ 0x70213a7b8	24	37,168
java.lang.String @ 0x70135d758	24	37,168
java.lang.String @ 0x700066bd8	24	32,112
java.lang.String @ 0x7021e3d78	24	26,224
java.lang.String @ 0x700f7e450	24	26,224
java.lang.String @ 0x702294c50	24	23,632
java.lang.String @ 0x701dde1c8	24	23,632
Total: 14 of 236,646 entries; 236,632 more		

图 7-6：`String` 对象保留的内存

大小固定的 Hashtable

如果尚不熟悉 Hashtable 和 Hashmap，你可能想知道到底什么是大小固定的 Hashtable（特别是，这些类的 Java 实现大小都是不固定的）。

从概念上讲，一个 Hashtable 包含一个数组，它会保存一些条目（数组中的每个元素叫作一个桶）。当要将一个对象保存到 Hashtable 中时，可以用该对象的哈希值对桶的数目取余，以此确定对象在数组中的存储位置。这种情况下，两个哈希值不同的对象很有可能被映射到同一个桶中，每个桶实际就是一个链表，其中按顺序存储了映射到该桶的条目。当两个对象映射到一个桶时，这就叫"冲突"。

随着越来越多的对象被插入到这个表中，冲突也会越来越多；进而会有更多的条目被插入到每个链表中。要找到一个条目，就变成了在一个链表中搜索。这可能会非常慢，特别是随着链表越来越长，速度会更慢。

解决方案是设置 Hashtable 的大小，以便它有更多的桶（当然，结果就是冲突会减少）。很多实现都是动态处理的；实际上，Java 的 `Hashtable` 和 `HashMap` 也是这么工作的。

其他实现，像这里讨论的 JVM 内部的这个，就不能重新设置 Hashtable 的大小；其数组的大小是在创建时就固定的。

其他实现，像这里讨论的 JVM 内部的这个，就不能重新设置 Hashtable 的大小；其数组的大小是在创建时就固定的。

从 Java 7 中开始，这个表的大小可以在 JVM 启动时使用 `-XX:StringTableSize=N`（如前面所介绍的，默认值为 1009 或 60 013）。如果某个应用会保留大量字符串，就应该增加这个值。如果这个值是个素数，字符串保留表的效率最高。

`intern()` 方法的性能是由表大小的调优程度所决定的。作为一个例子，表 7-3 列出了在不同场景下创建和保留 1 千万个随机创建的字符串的总时间：

表7-3：保留1千万个字符串的时间

调优	用时
字符串表大小为 1009	2.3 小时
字符串表大小为 1 百万	30.4 秒
字符串表大小为 1 千万	25.2 秒
自定义方式	26.4 秒

注意，如果字符串保留表的大小设置不当，性能损失会相当严重。一旦根据预期数据设置了该表的大小，性能会极大改善。

最后一个测试用例没有使用 `intern()` 方法，而是使用了前面介绍的示例 `canonicalVersion()` 方法，它是用 `CustomConcurrentHashMap` 类实现的（出自 JSR 166 的一个早期版本），而且用的是非强引用的键和值。与精心优化过的字符串保留表相比，这对性能没什么帮助。不过这种方案也有一个优势，即开发者根本不需要调节其大小。`CustomConcurrentHashMap` 的初始大小是 1009，它会根据需要动态调整大小。与最大程度优化过的字符串表大小相比，还是有比较小的性能损失，但是运行要容易得多。（不过在那种情况下，代码必须调用定制类的 `canonicalVersion()` 方法，而不是简单地替换掉 `intern()` 方法。）

如果想看看字符串表的执行过程，可以使用 `-XX:+PrintStringTableStatistics` 参数（这个标志要求 JDK 7u6 或更新版本，默认为 `false`）运行应用。当 JVM 退出时，它会打印一个这样的列表：

```
StringTable statistics:
Number of buckets       :    1009
Average bucket size     :    3008
Variance of bucket size :    2870
Std. dev. of bucket size:      54
Maximum bucket size     :    3186
```

这个命令行也会显示符号表的信息，但是这里我们感兴趣的是字符串表。（符号表用于保存一些类信息。JDK 8 有一个调整该表大小的实验性选项，但是一般不会调整它。）在这个例子中，有 3 035 072 个保留的字符串（因为有 1009 个桶，每个桶平均有 3008 个字符串）。理想情况下，桶的平均大小应该是 0 或 1。这个大小实际上不会为 0，可能会小于 0.5，但是因为计算时用的是整型运算，所以报告中会向下取整。如果平均值大于 1，则需要增大字符串表的大小。

某个应用中已经分配的保留字符串个数（及其总大小），可以使用如下的 `jmap` 命令获得

（这也需要 JDK 7u6 或更新版本）：

```
% jmap -heap process_id
…… 其他输入 ……
36361 interned Strings occupying 3247040 bytes.
```

如果将字符串表设得特别大，其损失是非常小的：每个桶只需要 4 字节或 8 字节（取决于使用的是 32 位还是 64 位 JVM），所以比最优的情况多几千，只是一次性消耗一些原生内存（不是堆内存）。

> **字符串的 Intern 和 Equals**
>
> 在谈到保留字符串这个主题时，因为保留的字符串可以通过 == 操作符比较，那使用 `intern()` 方法让程序跑得快一些怎么样呢？这种想法很常见，但是大部分情况下并非如此。`String.equals()` 方法是相当快的。首先要知道，长度不相等的字符串肯定不会相同；即使长度相同，还要扫描字符串，比较所有的字符（至少要找到不匹配的地方）。不可否认，通过 == 操作比较字符串确实会快一些，但是保留字符串的成本也要考虑进去。这需要（还有其他方面）计算字符串的哈希编码，这意味着要扫描整个字符串，并在每个字符上执行一个操作（就像 `equals()` 所做的那样）。
>
> 只有一种情况下会有好处：应用会在一组长度相同的字符串上执行大量的重复比较。如果字符串都已经保留了，那用 == 作比较更快；调用 `intern()` 的代价是只需要计算一次。但是一般而言，性能差不多。

快速小结

1. 如果应用中有大量字符串是一样的，那通过保留实现字符串重用收效很大。
2. 要保留很多字符串的应用可能需要调整字符串保留表的大小（除非是运行在 Java 7u40 及更新的 64 位服务器 JVM 上）。

7.3　对象生命周期管理

关于内存管理，本章要大篇幅讨论的第二个主题是对象生命周期管理。在很大程度上，Java 会尽量减轻开发者投入到对象生命周期管理上的精力：开发者在需要的时候创建对象，当不再需要这些对象时，它们会走出作用域，并由垃圾收集器释放。

有些情况下，正常的生命周期并不是最优的。有些对象创建的成本很高，而管理这些对象的生命周期可以改进应用的效率，即便以让垃圾收集器多做些工作为代价。本节将探索正常的生命周期何时应该有所改变，以及如何改变，手段可以是重用对象，或者是维护指向这些对象的特殊引用。

7.3.1　对象重用

对象重用通常有两种实现方式：对象池和线程局部变量。一说这个，开发 GC 的工程师就

要抱怨了，因为这两种技术都会影响 GC 的效率。特别是对象池，GC 圈对其是很抵触的，而且由于其他很多原因，开发圈也不太喜欢这种技术。

从某种程度上说，对象池技术之所以不受待见，原因似乎显而易见：被重用的对象会在堆中停留很长时间。如果有大量对象存在于堆中，那用来创建新对象的空间就少了，因为 GC 操作会更为频繁。不过这只是冰山一角。

第 6 章曾经介绍过，对象创建时是分配在 Eden 区的。在最终提升到老年代之前，会在 Survivor 区反复经历一些 Young GC 周期。每当处理到最近创建或者新创建的池化对象时，GC 算法必须执行一些工作，去复制这个对象，并调整指向它的引用，直到该对象最终进入老年代。

尽管看上去故事可能就此结束了，但是一旦对象被提升到老年代，可能引发的性能问题甚至会更多。执行一次 Full GC 所花的时间与老年代中仍然存活的对象数量成正比。存活对象的数量甚至比堆的大小更重要；处理一个 3 GB 大小但存活对象很少的老年代，与处理一个 1 GB 大小但存活对象占 75% 的老年代相比，速度要快一些。

GC 效率

那么堆中存活对象的数量对 GC 时间有多少影响呢？答案不一而足。

如下是在我的标准 4 核 Linux 系统上所做的一个测试的 GC 日志输出，测试中使用了 4 GB 的堆（其中 1 GB 固定为新生代）：

```
[Full GC [PSYoungGen: 786432K->786431K(917504K)]
        [ParOldGen: 3145727K->3145727K(3145728K)]
        3932159K->3932159K(4063232K)
        [PSPermGen: 2349K->2349K(21248K)], 0.5432730 secs]
        [Times: user=1.72 sys=0.01, real=0.54 secs]
...
[Full GC [PSYoungGen: 786432K->0K(917504K)]
        [ParOldGen: 3145727K->210K(3145728K)]
        3932159K->210K(4063232K)
        [PSPermGen: 2349K->2349K(21248K)], 0.0687770 secs]
        [Times: user=0.08 sys=0.00, real=0.07 secs]
...
[Full GC [PSYoungGen: 349567K->349567K(699072K)]
        [ParOldGen: 3145727K->3145727K(3145728K)]
        3495295K->3495295K(3844800K)
        [PSPermGen: 2349K->2349K(21248K)], 0.7228880 secs]
        [Times: user=2.41 sys=0.01, real=0.73 secs]
```

注意中间的输出：应用清理了指向老年代中的大部分引用，所以在 GC 之后，老年代中的数据只有 210 KB 了。该操作仅用了 70 毫秒。在其他情况下，堆中的大部分数据仍然存活；Full GC 操作尽管几乎没有移除什么数据，但是花费的时间分别是 540 毫秒和 730 毫秒。这还算是幸运的，测试中有 4 个 GC 线程。在单核系统上，这个例子中耗时较短的 GC 需要 80 毫秒，耗时较长的 GC 则需要 2410 毫秒（超过 30 倍的差距）。

使用某个并发收集器避免 Full GC 并不能使情况有所好转，这是因为，并发收集器的标记

阶段所花的时间也依赖于仍存活数据的数量。特别是对 CMS 而言，池中的对象很可能会在不同的时间被提升，这会增大因碎片而导致的并发故障的机会。总的来说，对象在堆中存留的时间越长，GC 的效率越差。

因此，对象重用并不好。现在我们可以讨论如何以及何时重用对象了。

JDK 提供了一些常见的对象池：线程池（将在第 9 章讨论）和软引用。软引用（本节后面会讨论）本质上是一大池可重用对象。同时 Java EE 依赖对象池来连接数据库和其他资源，而且 EJB（Enterprise Java Beans）的整个生命周期都是围绕对象池的概念构建的。

线程局部变量的情况类似；JDK 中到处是使用线程局部变量的类，以避免重新分配特定种类的对象。

显然，甚至 Java 专家都理解在某些情况下需要对象重用。

之所以要重用对象，原因是很多对象初始化的成本很高，与增加的 GC 时间这一点相权衡，重用更为高效。对于像 JDBC 连接池这样的东西，肯定如此：创建网络连接，以及可能还要进行的登录和建立数据库会话，成本非常高。这种情况下，对象池有很大的性能优势。线程也可以池化，以节省创建线程的时间；随机数生成器是作为线程局部变量提供的，以节省生成随机数的时间；诸如此类。

这些例子有一个共同的特性，即初始化对象需要的时间较长。在 Java 中，对象分配非常快，成本也不高（反对对象重用的观点往往就是关注的这一点）。对象初始化的性能取决于对象本身。应该只考虑重用初始化成本非常高的对象，而且是只有当初始化这些对象的代价在程序中是主导性操作之一时。

这些例子还有一个共性，那就是所共享对象的数目往往很小，以便最小化对 GC 的影响：即它们的数量较小，还不足以降低 GC 周期。池中有少量对象，对 GC 效率不会影响太大；如果堆中满是池化对象，就会严重影响 GC 了。

下面是 JDK 和 Java EE 中重用对象的一些例子，以及重用的原因：

线程池

  线程初始化的成本很高。

JDBC 池

  数据库连接初始化的成本很高。

EJB 池

  EJB 初始化的成本很高（参见第 10 章）。

大数组

  Java 要求，一个数组在分配的时候，其中的每个元素都必须初始化为某个默认值（`null`、`0` 或者 `false`，根据具体情况而定）。对于很大的数组，这是非常耗时的。

原生 NIO 缓冲区

  不管缓冲区多大，分配一个直接的 `java.nio.Buffer`（即调用 `allocateDirect()` 方法返回的缓冲区），这个操作都非常昂贵。最好是创建一个很大的缓冲区，然后通过按需切

割的方式来管理，以便将其重用于以后的操作。

安全相关类

`MessageDigest`、`Signature` 以及其他安全算法的实例，初始化的成本都很高。基于 Apache 的 XML 代码就是使用线程局部变量保存这些实例的。

字符串编解码器对象

JDK 中的很多类都会创建和重用这些对象。在大多数情况下，这些还是软引用，下一节将介绍。

`StringBuilder` 协助者

`BigDecimal` 类在计算中间结果时会重用一个 `StringBuilder` 对象。

随机数生成器

`Random` 类和（特别是）`SecureRandom` 类，生成它们的实例的代价是很高的。

从 DNS 查询到的名字

网络查询代价很高。

ZIP 编解码器

有一种有趣的变化，初始化的开销不是特别高，但是释放的成本很高，因为这些对象要依赖对象终结操作（finalization）来确保释放掉所用的原生内存。更多细节，参见 7.3.2 节的“终结器和最终引用”。

此处讨论的对象池和线程局部变量两种方式，在性能上有些差别。下面详细看一下。

1. 对象池

对象池不受人喜欢，原因有多个方面，只有部分原因和性能有关。线程池的大小可能很难正确地设置，它们将对象管理的负担又抛给程序员了：程序员不能简单地将对象丢出作用域，而必须记得将其返还到对象池中。

不过这里的焦点是对象池的性能，它受如下几个因素的影响。

GC 影响

如我们所见，保存大量对象会降低 GC 的效率（有时非常显著）。

同步

对象池必然是同步的，如果对象要频繁地移除和替换，对象池上可能会存在大量竞争。其结果是，访问对象池可能比初始化新对象还慢。

限流（Throttling）

对象池对性能也有正面的影响：对于对稀缺资源的访问，线程池可以起到限流作用。如第 2 章所讨论的，如果想增加的负载超出系统的处理能力，性能将下降。这是线程池之所以很重要的一个原因。如果有太多线程同时运行，CPU 将不堪重负，而且性能会下降（第 9 章就有个这方面的例子）。

这一原则也适用于远程系统的访问，而且在 JDBC 连接中会经常见到。如果 JDBC 连接数超出数据库的处理能力，数据库的性能就会下降。在这些情况下，通过确定池的上限

来限制资源数（如 JDBC 连接数）更好，即便这意味着应用中的线程必须等待一个空闲资源。

2. 线程局部变量

在通过将对象保存为线程局部变量这种技术实现对象重用时，有不同的性能权衡，如下所列。

生命周期管理

线程局部变量要比在池中管理对象更容易，成本更低。这两种技术都邀请开发者去获取初始对象：或者是从对象池中检出，或者是在线程局部对象上调用 `get()` 方法。但是对象池还要求开发者在使用完毕后归还对象（否则其他人就不能使用了）；线程局部对象在线程内总是可用的，不需要显式地归还。

基数性（Cardinality）

线程局部变量通常会伴生线程数与保存的可重用对象数之间的一一对应关系。不过并非严格如此。线程的变量副本，直到该线程第一次访问它时，才会创建，因此保存的对象数有可能小于线程数。但是保存的对象数不可能会超过线程数，大部分时间两者是相同的。

另一方面，对象池的大小则有些随意。如果一个 Servlet 有时需要一个 JDBC 连接，有时需要两个，则 JDBC 池的大小可以相应设定（比如说，对于 8 个线程，设定 12 个连接）。线程局部变量做不到这一点，也不能减少对资源的访问（除非线程数本身可以减少）。

同步

线程局部变量不需要同步，因为它们只能用于一个线程之内；而且线程局部的 `get()` 方法相当快。（情况并非一直如此，在早期的 Java 版本中，获得一个线程局部变量的开销很大。如果过去因为差劲的性能而远离了线程局部变量，在当前的 Java 版本中，可以重新考虑一下。）

同步还带来了一个有趣的问题，因为线程局部对象的性能优势通常会用节省了同步的代价来表达（而不说这是重用对象的好处）。比如，Java 7 引入了一个 `ThreadLocalRandom` 类；这个类（而不是一个 `Random` 实例）也用到了示例股票应用中。此外，本书中的很多例子在 `Random` 对象的 `next()` 方法上都会遇到一个同步瓶颈。使用线程局部对象是避免同步瓶颈的好方法，因为只有一个线程能使用这个对象。

然而，只要让这个例子每次需要时，就简单地创建一个新的 `Random` 实例，同步问题也能轻松解决。不过，这样解决同步问题对整体性能没什么帮助：初始化一个 `Random` 对象的开销非常大，而且持续创建这个类的实例，与在多个线程间共享一个类实例的同步瓶颈相比，性能可能更差。

使用 `ThreadLocalRandom` 类性能会更好，如表 7-4 所示。这个例子使用了 batching stock 应用，对于每支股票，有创建新的 `Random` 实例和重用 `ThreadLocalRandom` 两种方案。

表7-4：在计算股票时使用`ThreadLocalRandom`的效果

股票数量	分配新的Random（秒）	重用ThreadLocalRandom（秒）
1	0.174	0.175
10	0.258	0.236

（续）

股票数量	分配新的Random（秒）	重用ThreadLocalRandom（秒）
100	0.564	0.49
1000	2.308	1.916
10 000	17.32	13.55

对于一般的对象重用，这里的经验是，在初始化对象需要很长时间时，不用畏惧探索用对象池或线程局部变量技术来重用那些创建开销高昂的对象。不过还是要找到一个平衡点：对于一般的类，较大的对象池所带来的性能问题很可能比解决的问题还要多。所以应该将这些技术应用于初始化成本高昂，以及重用对象的数目比较小时。

快速小结

1. 对象重用通常是一种通用操作，我们并不鼓励使用它。但是这种技术可能适合初始化成本高昂，而且数量比较少的一组对象。
2. 在使用对象池还是使用线程局部变量这两种技术之间，应该有所取舍。一般而言，建设线程和可重用对象直接存在一一对应关系，则线程局部变量更容易使用。

7.3.2 弱引用、软引用与其他引用

在 Java 中，弱引用和软引用也支持对象重用，不过作为开发者，我们并不会经常从重用的角度看待它们。我会一般性地将其称作非确定引用。这些类引用更多用于缓存一个较长的计算或者一个数据库查询的结果，而非用于重用对象。比如，在股票 Servlet 中，可以用一个非确定引用来缓存 `getHistory()` 方法（该方法需要很长的计算，或者需要很长的数据库调用）的结果。这个结果只是一个对象，当通过非确定引用来缓存它时，我们只是简单地重用了该对象，不然的话，初始化开销会很高。

术语说明

讨论弱引用和软应用可能会令人困惑，因为很多术语使用了类似的词汇。下面是这些术语的一个简单入门介绍。

引用（Reference）

引用（或者说对象引用）可以是任何类型的引用：强引用、弱引用、软引用等。指向一个对象的普通引用实例变量就是一个强引用。

非确定引用（Indefinite reference）

本书使用这个术语来区分强引用和其他特殊引用（比如软引用或弱引用）。一个非确定应用其实是一个对象实例（比如，`SoftReference` 类的一个实例）。

所引对象（Referent）

非确定引用的工作方式是，在非确定引用类的实例内，嵌入另一个引用（几乎总是嵌入一个强引用）。被封装的对象称作“所引对象”。

不过，很多程序员仍然会有不同的感觉。实际上，该术语也反映出这样一点：没有人说“缓存”一个线程用于重用，但是我们将在缓存数据库操作结果方面探索非确定引用的重用。

与对象池或线程局部变量相比，非确定引用的优势在于，它们最终会被垃圾收集器回收。如果对象池中包含了已经执行的最后 10 000 个股票查询，堆的运行就会变慢，应用也会受牵连：去掉那 10 000 个元素所占据的堆，剩下的就是应用可以使用的其余堆了。如果这些查询是通过非确定引用保存的，JVM 就可以释放一些空间（取决于引用的类型），从而获得更好的 GC 吞吐量。

非确定引用的缺点是对垃圾收集器的效率会有轻微影响。图 7-7 对比了不使用与使用非确定引用时内存的使用情况（这里用的是弱引用）。

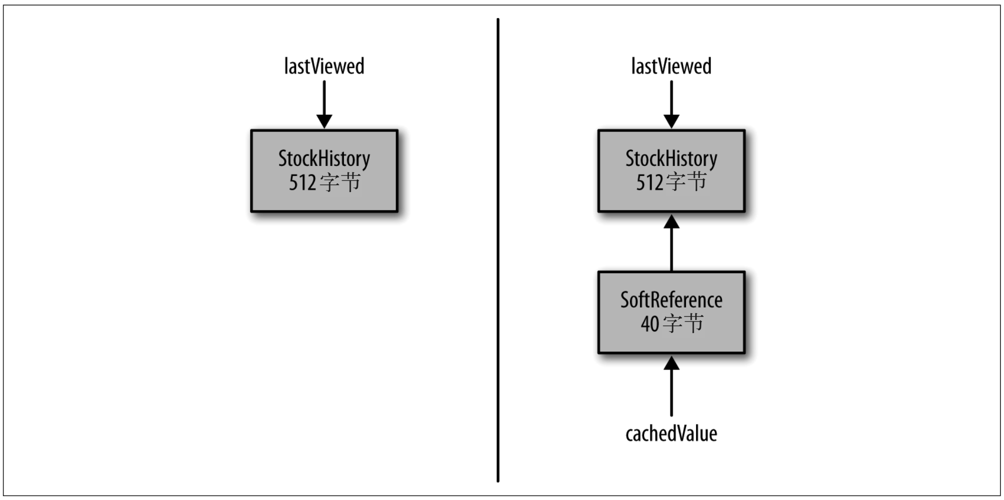


图 7-7：非确定引用的内存分配

被缓存的对象占了 512 字节。在左侧的就是消耗的所有内存（没有指向对象的实例变量占据的内存）。在右侧，对象被缓存在一个 `SoftReference` 内，额外增加了 40 字节的内存消耗。非确定引用和其他任何对象一样：它们也消耗内存，而且其他变量（图中右侧的 `cachedValue` 变量）也是通过强引用引用它们。

所以对垃圾收集器的第一个影响是，非确定引用会导致应用使用更多内存。对垃圾收集器的更大的影响体现为，垃圾收集器要回收非确定引用，至少需要两个 GC 周期。

图 7-8 说明了当一个所引对象不再被强引用时（即 `lastViewed` 被设置为 `null`），会发生什么。如果没有对 `StockHistory` 对象的引用，在下一次 GC 期间，该对象会被释放。所以图的左侧现在消耗的内存为 0 字节。

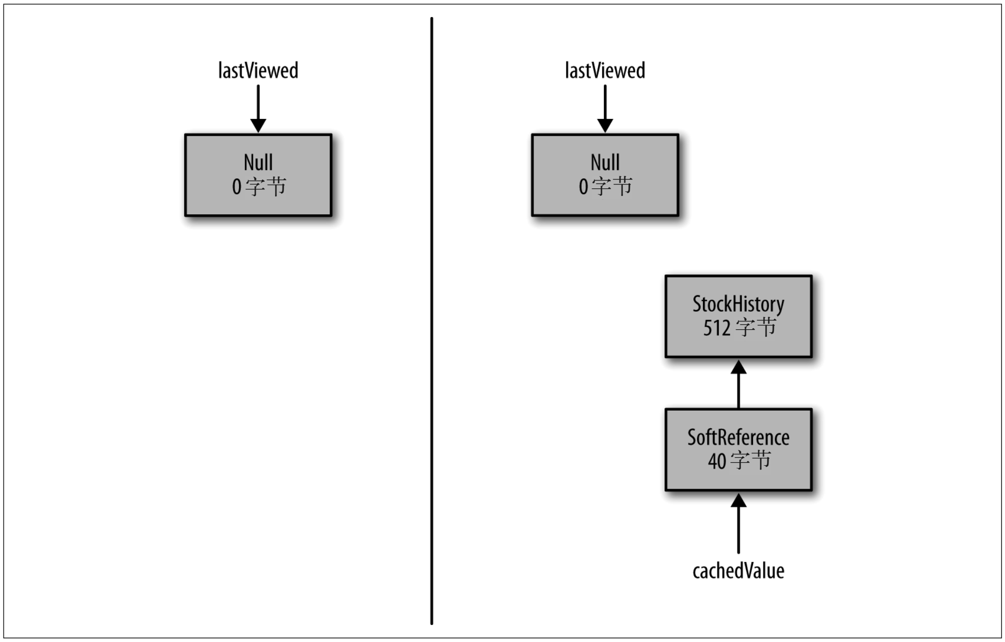


图 7-8：在 GC 周期间非确定引用保留的内存

在图的右侧，仍然有内存消耗。所引对象被释放的精确时机，会随非确定引用类型的不同而有所不同，暂时只考虑软引用的情况。所引对象将仍然逗留在内存中，直到 JVM 确定近期不会再使用它。当这个条件出现时，第一次 GC 会释放所引对象，但不是非确定引用本身。应用最终的内存状态如图 7-9 所示。

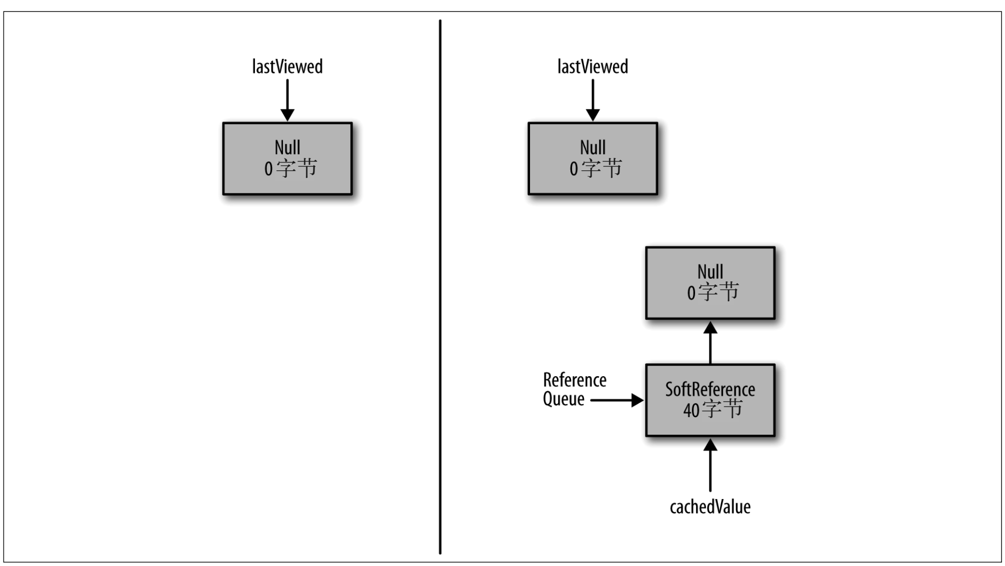


图 7-9：非确定引用不会立即清理

现在，对于非确定引用对象本身，（至少）有两个强引用指向它：由应用创建的原始的强引用，再就是由 JVM 创建的、在所引对象队列上的一个新的强引用。在非确定引用对象本身被垃圾收集器回收之前，必须先清理掉所有这些强引用。

这种代码通常是由处理引用队列的代码处理的。如果在队列上有新对象创建，代码会得到通知，并立即移除指向该对象的所有强引用。之后，在下一个 GC 期间，非确定引用对象会被释放。最糟糕的情况是，引用队列没有立即被处理，有可能要经过多个 GC 周期，才能将一切清理干净。然而即便在最好的情况下，非确定引用在释放之前也必须经历两个 GC 周期。

依赖于非确定引用的类型，处理算法也有较大差异，但是所有的非确定引用某种程度上都有这类性能损失。

GC 日志与引用处理

当运行一个使用了大量非确定引用的对象时，可以考虑添加 -XX:+PrintReferenceGC 标志（默认为 false）。这样就能看到处理这些引用花了多少时间：

```
[GC[SoftReference, 0 refs, 0.0000060 secs]
        [WeakReference, 238425 refs, 0.0236510 secs]
        [FinalReference, 4 refs, 0.0000160 secs]
        [PhantomReference, 0 refs, 0.0000010 secs]
        [JNI Weak Reference, 0.0000020 secs]
        [PSYoungGen: 271630K->17566K(305856K)]
        271630K->17566K(1004928K), 0.0797140 secs]
        [Times: user=0.16 sys=0.01, real=0.08 secs]
```

在这个例子中，238 425 个弱引用的使用使得 Young GC 的时间增加了 23 毫秒。

1. 软引用

如果问题中的对象以后有很大的机会重用，可以使用软引用，但是如果该对象近期一直没有使用到（计算时也会考虑堆还有多少内存可用），垃圾收集器会回收它。软引用本质上是一个比较大的、最近最久未用（LRU）的对象池。获得较好性能的关键是确保它们会被及时清理。

来看一个例子。股票 Servlet 可以设置一个股票历史的全局缓存，以股票代码（或者代码与日期）为键。比如有请求要获取 TPKS 从 2013 年 6 月 1 日到 2013 年 8 月 31 日之间的股价历史，可以先看看缓存，其中是不是有以前类似请求的结果。

之所以要缓存数据，原因是对某类数据的请求往往会比其他数据更多。如果对 TPKS 这支股票的请求最多，就可以考虑将其保存到软引用缓存中。另一方面，查询一次 KENG 这支股票，其结果也会在缓存中停留一段时间，但最终会被回收。对于随时间变化的请求，也是如此：对 DNLD 的一群请求，可以利用第一次请求的结果。如果用户意识到 DNLD 是笔糟糕的投资，那些缓存的条目最终会从堆中去掉。

精确地讲，一个软引用何时会被释放呢？首先，所引对象一定不能有其他的强引用。如果软引用是指向其所引对象的唯一引用，而且该软引用最近没有被访问过，则所引对象会在下一次 GC 周期释放。具体而言，其关系可以用如下伪代码表示：

```
long ms = SoftRefLRUPolicyMSPerMB * AmountOfFreeMemoryInMB;
if (now - last_access_to_reference > ms)
   free the reference
```

这里有两个关键值。第一个是由 `-XX:SoftReflRUPolicyMSPerMB=N` 标志设置的，默认值为 1000。

第二个是堆中空闲内存的数量（在一个 GC 周期完成之后）。因为堆的大小是动态变化的，在计算堆中有多少内存可用时，JVM 有两个选择：堆中目前的空闲内存数量，或者堆扩展到最大容量后的空闲内存数量。这些值的选择是由所用的编译器确定的。client 编译器是基于当前堆中的可用值，而 server 编译器堆的最大可能值。

那这都是怎么工作的呢？以使用 server 编译器、堆空间为 4 GB 的 JVM 为例，在一次 Full GC（或一个并发周期）之后，堆可能被占用了 50%，因此空闲堆是 2 GB。`SoftReflRUPolicyMSPerMB` 的默认值（1000）意味着在过去的 2048 秒（2 048 000 毫秒）内没有访问到的任何软引用都会被清理：空闲堆是 2048（MB），再乘以 1000：

```
long ms = 2048000; // 1000 * 2048
if (System.currentTimeMillis() - last_access_to_reference_in_ms > ms)
    free the reference
```

如果 4 GB 的堆占用了 75%，则过去的 1024 秒内没有访问到的对象会被回收，以此类推。

要更频繁地回收软引用，可以降低 `SoftReflRUPolicyMSPerMB` 标志的值。将该值设置为 500，意味着堆大小为 4 GB 的 JVM 如果占用了 75%，则会回收过去 512 秒没有访问到的对象。

如果堆很快就会被软引用填满，则调优该标志往往是必要的。假设堆有 2 GB 空闲，应用开始创建软引用。如果它在不到 2048 秒（大概是 34 分钟）创建了 1.7 GB 的软引用，则这些软引用都不满足回收条件。这样，堆中留给其他对象的空间就只有 300 MB 了；这会导致 GC 频繁进行（对整体性能影响很坏）。

如果 JVM 完全耗尽了内存，则会出现非常严重的颠簸（thrashing），它会清理掉所有的软引用，否则会抛出 `OutOfMemoryError`。不抛出错误当然好，但是不分青红皂白地丢掉所有缓存的结果，可能也不理想。因此，另一个降低 `SoftReflRUPolicyMSPerMB` 的值的时机是，当引用处理日志说明有大量软引用意外被清理时。如 7.1.3 节的“达到 GC 的开销限制”一节所讨论的，这种情况在 4 次连续的 Full GC 周期之后才会发生（而且仅当其他因素都已满足时）。

另一方面，对于长期运行的应用，如果满足如下两个条件，可以考虑增大 `SoftReflRUPolicyMSPerMB` 的值：

- 有很多空闲堆可用；
- 软引用会频繁访问。

这种情况非常罕见。这与设置 GC 策略所讨论的情况类似：可以想象，如果增加了软引用策略的值，就是告诉 JVM，不到万不得已不要释放软引用。确实如此，但是这同时也告诉 JVM，堆中不要给正常操作留任何空间，结果很可能会导致把很多时间花在 GC 上。

应该注意的是，不要使用太多软引用，因为它们很容易填满整个堆。与提防创建包含太多实例的对象池相比，这一点更应该注意：如果对象的数目不是特别大，软引用就会工作得很好。否则，就要考虑用更传统的、固定大小的对象池来实现一个 LRU 缓存。

2. 弱引用

当问题中的所引对象会同时被几个线程使用时，应该考虑弱引用。否则，弱引用很可能会被垃圾收集器回收：只有弱引用的对象在每个 GC 周期都可以回收。

这意味着，弱引用绝对不会进入图 7-8 所示的软引用的状态。当强引用被移除时，弱引用会立即释放。因此程序的状态直接从图 7-7 到达图 7-9。

这里有个有趣的现象，弱引用会在堆中终结。引用对象就和其他 Java 对象一样：在年轻代中创建，最终会被提升到老年代。如果弱引用本身仍然在年轻代中，而弱引用的所引对象被释放了，则弱引用可以快速释放（下一次 Minor GC 时）。（假定问题中对象的引用队列会被快速处理。）如果弱引用的所引对象存在了足够长的时间，被提升到了老年代中，则弱引用在下一次并发或 Full GC 周期内才会释放。

以股票 Servlet 的缓存为例，假设我们知道某个特定用户会在其会话期间访问 TPKS，他几乎总会再次访问。在该用户的 HTTP 会话中，用一个强引用来保存股票的值是有意义的：它会一直存在，一旦用户登出，HTTP 会话就会被清理，而内存也会被回收。

如果另一个用户来了，而且也需要 TPKS 的数据，那如何找到数据呢？因为对象在内存中的某个地方，我们不想重新去查找，但是 Servlet 代码不能搜索其他用户的会话数据，因此，除了在第一个用户的 HTTP 会话中保存一个指向 TPKS 数据的强引用外，在一个全局缓存中保存一个弱引用，指向那个数据，也是有意义的。现在第二个用户就能查找 TPKS 数据了，当然这是以第一个用户没有登出并清理会话为前提的。（这就是本章的堆分析一节中所用的场景，其中的数据有两个引用，通过在最大的保留内存中查看对象，并不容易查找。）

这就是所谓的同时访问。这就好比是告诉 JVM：“嘿，只要有其他人对这个对象感兴趣，就让我知道它在哪儿，但是如果他们不再需要它了，就把它丢弃，我自己会重新创建。”比较弱引用与软引用，软引用基本像是在说：“嘿，只要有足够的内存，而且看上去有人会偶尔访问它，那就保留着它。”

如果不理解这种区别，在使用弱引用时就经常会出现性能问题。不要认为除了释放更快，弱引用和软引用就是一样的，别犯这种错误：软引用的对象通常可以存活几分钟甚至几小时，但是只要所引对象仍然存在，弱引用对象就一直存活。（下一个 GC 周期会清理。）

非确定引用与集合

在 Java 中，集合类经常是内存泄漏的根源：比如，某个应用将对象放入了一个 HashMap 对象中，却从不移除。随着时间的推移，这个 HashMap 对象就会越来越大，而且消耗堆。

为处理这种情况，开发者喜欢的一种方式是使用保存非确定引用的集合类。JDK 提供了两个这样的类：`WeakHashMap` 和 `WeakIdentityMap`。很多第三方库中都有基于软引用（即其他引用）定制集合类的用法（包括 JSR 166 的示例实现，比如用在如何创建和保存 canonical 对象的例子中使用的那个）。

使用这些类很方便，但是注意，它们有两大开销。其一，如本节所讨论的，非确定引用对垃圾收集器有不利影响。其二，类本身必须周期性地执行一个操作，清理集合中所有的未引用数据（也就是说，这个类负责处理它所保存的非确定引用的引用队列）。

例如，`WeakHashMap` 类的键使用了弱引用。当弱引用的键不再可用时，`WeakHashMap` 代码必须清理掉其中与该键关联的值。每次引用到这个映射时，都会执行该操作：处理弱键的引用队列，从映射中移除与引用队列中的任何键关联的值。

性能方面，有两点意义。第一，弱引用及其关联值，当这个映射再一次被用到时，才会实际释放。因此，如果这个映射使用不是很频繁，则意味着与映射关联的内存不会如预期般释放得那么快。

第二，它意味着该映射上的操作的性能是难以预测的。正常而言，hashmap 上的操作非常快；这也是 hashmap 得以流行的原因所在。紧接在一次 GC 之后的 `WeakHashMap` 上的操作，必须处理引用队列；该操作的时间不再固定，而且可能会比较长。因此，即便键释放不是很频繁，性能还是很难预测。更糟的是，如果映射中的键会频繁释放，`WeakHashMap` 的性能可能会非常差。

基于非确定引用的集合可能很有用，但是应该谨慎使用。如果可能，让应用自己管理集合。

3. 终结器（Finalizer）和最终引用（Final Reference）

每个 Java 类都有一个从 `Object` 类继承而来的 `finalize()` 方法；在对象可以被垃圾收集器回收时，可以用这个方法来清理数据。这听上去是个不错的特性，而且在有些情况下是需要的。然而在实践中，结果往往很糟，应该尽量不要使用这个方法。

因为功能方面的原因，终结器不是很好，另外，它们的性能也不好。终结器实际上是非确定引用的一种特殊情况：JVM 使用了一个私有的引用类（`java.lang.ref.finalizer`，它又是 `java.lang.ref.finalReference` 的子类）来记录定义了 `finalize()` 方法的对象。当一个具有 `finalize()` 方法的对象被分配时，JVM 会分配两个对象：一个是该对象本身，另一个是一个以该对象为所引对象的 `finalizer` 引用。

和其他非确定引用一样，在非确定应用对象释放之前，至少需要两个 GC 周期。然而，这里的性能损失要比其他非确定引用类型大得多。当软引用或弱引用的所引对象可以被 GC 回收时，所引对象本身会立即释放；这就会出现前面图 7-9 所示的内存使用情况。弱引用或软引用放在引用队列中，但是引用对象不会再指向任何东西（也就是说，`get()` 方法会返回 `null`，而不是原来的所引对象）。在软引用和弱引用的情况下，两个周期的性能损失只是引用对象本身的（而非所引对象的）。

`finalReference` 就不是这样了。要调用所引对象的 `finalize()` 方法，`finalizer` 类的实现

必须能够访问该对象，因此，当终结器引用被放到引用队列中时，所引对象不能释放。当某个终结器的所引对象可以回收时，程序状态可以用图 7-10 表示。

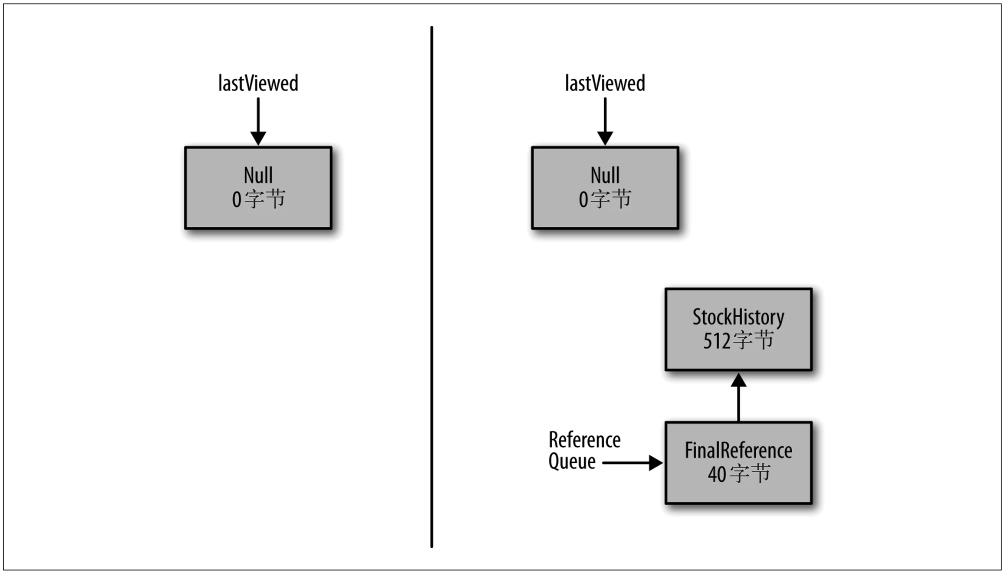


图 7-10：终结器引用要保留更多内存

当引用队列处理终结器时，`finalizer` 对象照例会被从队列中移除，之后就可以回收了。直到那时，所引对象才会被释放。与其他非确定引用相比，终结器对 GC 的性能影响更大，原因就在于此：与非确定引用对象本身消耗的内存相比，所引对象消耗的内存更为显著。

这会带来一个功能性问题，`finalize()` 方法可能会不小心又创建了一个指向所引对象的新的强引用。而这又会引起 GC 性能损失：现在，直到不再存在强引用时，所引对象才能释放。从功能上讲，这引发了一个大问题，因为当下一次所引对象可以被回收时，其 `finalize()` 方法不会被调用，预期的清理工作也不会进行。这类错误就足以解释为什么要尽可能少用终结器了。

遗憾的是，某些情况下终结器是不可避免的。比如，JDK 就在其操作 ZIP 文件的类中使用了终结器，因为打开 ZIP 文件会使用一些分配原生内存的原生代码。这些内存会在 ZIP 文件关闭时释放，但是如果开发者忘记调用 `close()` 方法，那该怎么办呢？事实上，终结器可以确保 `close()` 方法被调用，即便开发者忘了。

通常，如果使用终结器是不可避免的，那么一定要确保尽量减少该对象访问的内存。

对于使用终结器，还有一种替代方案，至少可以避免部分问题。特别是，这种方案支持在正常的 GC 操作期间释放所引对象。这是通过使用另一种非确定引用实现的，而非隐式地使用 `finalizer` 引用。

对于这个目的，有时推荐使用另一种非确定引用类型：`PhantomReference`（虚引用）类。这是个不错的选择，因为一旦没有指向所引对象的强引用了，引用对象就可以相当快地清

理，而且在调试的时候，该引用的意图会很清晰。当然也可以利用弱引用实现同样目标（另外，弱引用可以用在更多地方）。在特定情况下，如果软引用的缓存语义匹配应用的需求，可以使用软引用。

下面看一种替代方案。要创建一个替代的终结器，先创建非确定引用的一个子类，来保存需要在所引对象被回收后再清理的任何信息。然后在引用对象的一个方法内执行清理操作（与在所引对象内定义 finalize() 方法完全不同）。

```
private static class CleanupFinalizer extends WeakReference {

    private static ReferenceQueue<CleanupFinalizer> finRefQueue;
    private static HashSet<CleanupFinalizer> pendingRefs = new HashSet<>();

    private boolean closed = false;

    public CleanupFinalizer(Object o) {
        super(o, finRefQueue);
        allocateNative();
        pendingRefs.add(this);
    }

    public void setClosed() {
        closed = true;
        doNativeCleanup();
    }

    public void cleanup() {
        if (!closed) {
            doNativeCleanup();
        }
    }

    private native void allocateNative();
    private native void doNativeCleanup();
}
```

以上就是这样一个类的大概轮廓，它使用了一个弱引用。构造器中会分配一些原生资源。在正常使用的情况下，它会调用 setClosed() 方法，并清理原生内存。

不过这个弱引用也被放到了一个引用队列中。当该引用被从队列中取出时，可以检查原生内存是否已经清理，如果没有，就清理掉。

对引用队列的处理在一个守护线程中进行：

```
static {
    finRefQueue = new ReferenceQueue<>();
    Runnable r = new Runnable() {
        public void run() {
            CleanupFinalizer fr;
            while (true) {
                try {
                    fr = (CleanupFinalizer) finRefQueue.remove();
                    fr.cleanup();
                    pendingRefs.remove(fr);
```

```
                } catch (Exception ex) {
                    Logger.getLogger(
                            CleanupFinalizer.class.getName()).
                            log(Level.SEVERE, null, ex);
                }
            }
        }
    };
    Thread t = new Thread(r);
    t.setDaemon(true);
    t.start();
}
```

这些都放到一个私有的静态内部类中，隐藏到开发者使用的实际类之内，最后看上去是这样的：

```
public class CleanupExample {
    private CleanupFinalizer cf;
    private HashMap data = new HashMap();

    public CleanupExample() {
        cf = new CleanupFinalizer(this);
    }

    ……向hashmap中填东西的方法……

    public void close() {
        data = null;
        cf.setClosed();
    }
}
```

开发者就和构建其他任何对象一样构建这样的对象。开发者被告知要调用 `close()` 方法，它会清理原生内存，但是如果开发者没调用，也没问题。弱引用仍然存在于背后，所以当内部类处理弱引用时，`Cleanupfinalizer` 类自有机会处理原生内存。

这个例子中有一个技巧，即用 `pendingRefs` 保存弱引用。如果没有它，弱引用本身在有机会进入引用队列之前就会被收集了。

这个例子克服了传统终结器的两个局限性：它性能更好，因为所引对象一回收，与其关联的内存（这个例子中是 `data` 这个 `HashMap`）就会释放（而不是在 `finalizer()` 方法中进行）。而且所引对象是无法在清理代码中复活的，因为它已经被回收。

当然，其他反对使用终结器的意见也可以放到这里：我们无法确保垃圾收集器能抽出时间释放所引对象，也无法确保引用队列线程会处理队列中的任何特定对象。如果有很多这样的对象，处理引用队列的成本就会很高。和所有非确定引用一样，这个例子也应该谨慎使用。

终结器队列

终结器队列是一个引用队列，用于当所引对象可以被 GC 回收时处理 Finalizer 引用。

在执行堆转储分析时，确保终结器队列中没有对象，往往会方便一些：反正这类对象即将被释放，所以在堆转储中去掉它们，分析堆中的其他状况会更方便。可以通过如下命令让 JVM 处理终结器队列：

```
% jcmd process_id GC.run_finalization
```

要监控 Finalizer 队列，看看它是否是应用中的问题，可以在 jconsole 的 VM Summary 选项卡中看看它的大小（这是实时更新的）。脚本可以通过运行如下命令收集该信息：

```
% jmap -finalizerinfo process_id
```

快速小结

1. 非确定引用（包括软引用、弱引用、虚引用和最终引用）会改变 Java 对象正常的生命周期，与池或线程局部变量相比，它可以以对 GC 更为友好的方式实现对象重用。
2. 当应用对某个对象感兴趣，而且该对象在应用中的其他地方有强引用时，才应该使用弱引用。
3. 软引用保存可能长期存在的对象，提供了一个简单的、对 GC 友好的 LRU 缓存。
4. 非确定引用自身会消耗内存，而且会长时间抓住其他对象的内存；应该谨慎使用。

7.4 小结

内存管理对 Java 程序的快慢至关重要。调优 GC 非常重要，但是要获得最好的性能，在应用内必须有效地利用内存。

目前的硬件趋势往往不鼓励开发者考虑内存：如果我的笔记本有 16 GB 内存，我为什么要关心某个对象中有一个多余的、未使用的 8 字节大的对象引用呢？我们还忘记了一点，编程中通常的时间和空间之间的取舍，有可能会变成时间和空间与时间（time/space-and-time）之间的取舍：使用太多堆空间可能会降低性能，因为需要更多 GC。在 Java 中，管理堆仍然非常重要。

多数管理问题都围绕何时以及如何使用特殊的内存技术展开：对象池、线程局部变量和非确定引用。明智地使用这些技术可以极大改进应用性能，但是过度使用也很容易引起性能下降。在限量的情况下，也就是问题中的对象数目很少，而且有个边界时，使用这些内存技术会非常高效。

第 8 章

原生内存最佳实践

在 Java 应用中，堆消耗的内存是最多的。但是除堆之外，JVM 还会分配并使用大量的原生内存。第 7 章从编程的角度探讨了高效管理堆的不同方式，不过，堆的配置以及堆如何与操作系统的原生内存交互，是影响应用程序整体性能的另一个重要因素。

本章将从几个方面探讨原生内存（或者说操作系统内存）。我们将从 JVM 整体内存使用情况入手，目的在于理解如何监控内存的使用情况，以解决性能问题。之后将讨论为达到最理想的内存使用状况而采用的调优 JVM 和操作系统的不同方式。

8.1 内存占用

在 JVM 使用的内存中，通常堆消耗的部分最多，但是 JVM 也会为内部操作分配一些内存。这类非堆内存就是原生内存。应用中也可以分配原生内存（通过 JNI 调用 `malloc()` 和类似方法，或者是使用 New I/O，即 NIO 时）。JVM 使用的原生内存和堆内存的总量，就是一个应用总的内存占用（Footprint）。

从操作系统的视角看，总的内存占用是性能的关键。如果没有足够的物理内存来容纳应用总的内存占用，性能可能就要出问题了。这里有个词很关键："可能"。部分原生内存只是在启动时使用一下（比如与加载 classpath 下的 JAR 文件有关的内存），如果这类内存被交换出去，我们未必会注意到。有时候，一个 Java 进程使用的原生内存会与系统中的其他 Java 进程共享，还有更少一部分内存会与系统中的其他类型的进程共享。不过多数情况下，为优化性能，我们希望确保所有 Java 进程总的内存占用不超过机器的物理内存（加之可能还要给其他应用留些内存）。

8.1.1 测量内存占用

要测量一个进程总的内存占用，需要根据所用的操作系统选择特定的工具。在基于 Unix 的系统中，像 `top` 和 `ps` 这样的程序可以给出基本数据；在 Windows 中，可以使用 `perfmon` 或 `VMMap`。不管使用何种工具，何种平台，都需要看看进程实际分配的内存（这与保留的内存完全不同）。

之所以存在已分配内存和保留内存之分，是由 JVM（及所有程序）管理内存的方式导致的。考虑一个使用参数 `-Xms512m -Xmx2048m` 指定的堆，它一开始会使用 512 MB 的内存，之后会根据需要重新调整大小，以满足应用程序的 GC 目标。

这个概念就是提交内存（或者说已分配内存）和保留内存（有时叫作进程的虚拟内存）的本质区别。JVM 必须告知操作系统，它的堆可能需要多达 2 GB 的内存，所以会保留这么多内存：操作系统承诺，当 JVM 因为要增加堆而尝试分配额外的内存时，这些内存是可以获取到的。

最初分配的内存仍然是只有 512 MB，而且这就是堆实际用到的全部内存。这些已经实际分配的内存，就是提交内存。提交内存的量会随堆的重新调整而波动；特别是，提交内存会随着堆的增加而相应增加。

超量保留有没有问题？

在考察性能时，只有提交的内存才有价值，绝对不会因为保留了太多内存而出现性能问题。

不过有时还是需要确保 JVM 没有保留太多内存。32 位的 JVM 尤是如此。因为 32 位应用的最大进程空间是 4 GB（或者更少，跟操作系统有关），保留过多内存可能会成为问题。如果 JVM 为堆保留了 3.5 GB 的内存，那为栈、代码缓存等部分留下的原生内存就只有 0.5 GB 了。堆是不是只提交了 1 GB 内存并不重要，因为它保留了 3.5 GB，那给其他操作留下的内存就限制为 0.5 GB 了。

64 位的 JVM 没有进程空间大小的这种限制，但是又受限于机器的虚拟内存总量。比如说有一台小型服务器，物理内存有 4 GB，虚拟内存有 10 GB，我们启动一个堆大小为 6 GB 的 JVM。它会保留 6 GB 的虚拟内存（外加一些非堆内存部分）。不管这个堆实际增长到多大（提交多少内存），这台机器上的第二个 JVM 保留的内存都要小于 4 GB。

凡事都有两面，给 JVM 的内部结构多分配些空间，让 JVM 优化其使用，这样比较方便，但未必总是可行。

这种差异几乎存在于 JVM 分配的所有重要内存区域中。随着越来越多的代码被编译，代码缓存会从初始值向最大值增长。单独分配的持久代或元空间也会从初始大小（提交内存）向最大大小（保留内存）增长。

线程栈是个例外。JVM 每次创建线程时，操作系统会分配一些原生内存来保存线程栈，向进程提交更多内存（至少要等到线程退出）。线程栈是在创建时全部分配的。

在 Unix 系统中，一个应用实际的内存占用，可以用各种操作系统工具所报告的进程驻留集大小（Resident Set size，RSS）来估算。在评估一个进程使用的提交内存量时，这个值不失为一个好的衡量依据，不过它有两个不够精确的地方。其一，在 JVM 和其他进程之间，有些在操作系统层面共享的页面（共享库的 text 部分），会被计算在每个进程的 RSS 中。其二，随时可能会出现这样的情况，即一个进程的提交内存多于实际调入的页面。即便如此，跟踪一个进程的 RSS 仍是监控整体内存使用情况的不错的第一步。在较新的 Linux 内核中，PSS 是对 RSS 的改进，去掉了和其他程序共享的数据。

在 Windows 系统中，与 Unix 中的 RSS 等同的概念叫作应用的“工作集”（working set），这个信息是任务管理器提供的。

8.1.2 内存占用最小化

为将 JVM 的内存占用最小化，应该限制以下几个部分的内存使用量。

堆

　堆是最大的一块内存，尽管有些出人意料，它可能只占总内存占用的 50% 到 60%。可以将堆的最大值设置为一个较小的值（或者设置 GC 调优参数，比如控制堆不会被完全占满），以此限制程序的内存占用。

线程栈

　线程栈非常大，特别是对 64 位 JVM 而言。第 9 章会探讨限制线程栈消耗的内存量的不同方式。

代码缓存

　代码缓存使用原生内存来保存编译后的代码。第 4 章讨论过，代码缓存也可以调优（不过，如果因为空间的限制而导致所有代码无法编译，对性能也会有不利影响）。

直接字节缓冲区

　将在 8.1.3 节讨论。

8.1.3 原生NIO缓冲区

开发者可以通过 JNI 调用来分配原生内存，但是如果 NIO 字节缓冲区是通过 `allocateDirect()` 方法创建的，则也会分配原生内存。从性能的角度看，原生字节缓冲区非常重要，因为它们支持原生代码和 Java 代码在不复制的情况下共享数据。最常见的例子是用于文件系统和套接字（socket）操作的缓冲区。把数据写入一个原生 NIO 缓冲区，然后再发送给通道（channel，比如文件或套接字），不需要在 JVM 和用于传输数据的 C 库之间复制数据。如果使用的是堆字节缓冲区，JVM 则必须复制该缓冲区的内容。

调用 `allocateDirect()` 方法非常昂贵，所以应该尽可能重用直接字节缓冲区。理想的情况是，线程是独立的，而且每个线程持有一个直接字节缓冲区作为线程局部变量。如果有很多线程需要大小不同的缓冲区，有时可能会消耗过多原生内存，因为每个线程的缓冲区最终可能会达到最大值。对于这种情况，或者应用的设计不适合使用线程局部缓冲区时，直接字节缓冲区的对象池可能更有用。

字节缓冲区也可以切割管理。应用可以分配一个非常大的直接字节缓冲区，然后每个请求使用 `ByteBuffer` 类的 `slice()` 方法从中分配一部分。如果不能保证每次分配相同的大小，这种方案就很难处理：就像在分配和释放不同大小的对象时堆会呈现出碎片化一样，最初分配的这个字节缓冲区也会变得碎片化。然而与堆不同的是，字节缓冲区的不同片段是无法压缩的，所以只有当所有片段大小都相同时，这种解决方案才好用。

从调优的角度看，有一点需要知道，即不管使用上述哪种编程模型，应用可以分配的直接字节缓冲区的量都可以通过 JVM 加以限制。直接字节缓冲区所分配的内存总量，可以通过设置 `-XX:MaxDirectMemorySize=`*`N`* 标志来指定。从 Java 7 开始，这个标志的默认值为 0，这意味着没有限制（当然还是要受制于地址空间大小，以及操作系统对进程的各种限制）。可以使用这个标志来限制应用中直接字节缓冲区的使用（还可以利用它实现与 Java 以前版本的兼容，早期版本中，这个限制是 64 MB）。

快速小结

1. JVM 总的内存占用对性能影响很大，特别是当机器上的物理内存有限时。在做性能测试时，内存占用通常应该是要监控的一个方面。
2. 从调优角度看，要控制 JVM 的内存占用，可以限制用于直接字节缓冲区、线程栈和代码缓存的原生内存（以及堆）的使用量。

8.1.4 原生内存跟踪

从 Java 8 开始，借助 `-XX:NativeMemoryTracking=`*`off|summary|detail`* 这个选项，JVM 支持我们一窥它是如何分配原生内存的。原生内存跟踪（Native Memory Tracking，NMT）默认是关闭的（`off` 模式）。如果开启了概要模式（`summary`）或详情模式（`detail`），可以随时通过 `jcmd` 命令获得原生内存的信息：

```
% jcmd process_id VM.native_memory summary
```

如果 JVM 是使用 `-XX:+PrintNMTStatistics` 参数（默认为 `false`）启动的，它会在程序退出时打印原生内存分配信息。如下是一个初始堆大小为 512 MB，最大为 4 GB 的 JVM 的概要输出：

```
Native Memory Tracking:

 Total:  reserved=4787210KB,  committed=1857677KB
```

尽管 JVM 保留了总计 4.7 GB 的内存，但使用量远小于这个值——只有 1.8 GB。这非常典型（之所以没有特别注意 OS 工具中显示的进程虚拟大小，原因之一是它反映的只是保留内存）。

内存使用情况可以分解成如下几个部分：

```
-                 Java Heap (reserved=4296704KB, committed=1470428KB)
                            (mmap: reserved=4296704KB, committed=1470428KB)
```

不出所料，堆本身是 4 GB 保留内存中最大的一部分。但是堆的动态大小意味着它仅增长到了 1.4 GB。

```
-                    Class (reserved=65817KB, committed=60065KB)
                           (classes #19378)
                           (malloc=6425KB, #14245)
                           (mmap: reserved=59392KB, committed=53640KB)
```

这是用于保存类的元数据的原生内存。再次提醒，与实际用于保存程序中的 19 378 个类而占用的内存相比，JVM 保留的内存要更多。

```
-                   Thread (reserved=84455KB, committed=84455KB)
                           (thread #77)
                           (stack: reserved=79156KB, committed=79156KB)
                           (malloc=243KB, #314)
                           (arena=5056KB, #154)
```

77 个线程栈，每个分配了大约 1 MB 的空间。

```
-                     Code (reserved=102581KB, committed=15221KB)
                           (malloc=2741KB, #4520)
                           (mmap: reserved=99840KB, committed=12480KB)
```

这是 JIT 的代码缓存：19 378 个类并不是非常多，所以提交的代码缓存只是很小的一部分。

```
-                       GC (reserved=183268KB, committed=173156KB)
                           (malloc=5768KB, #110)
                           (mmap: reserved=177500KB, committed=167388KB)
```

这是 GC 算法的处理所使用的一些堆外空间。

```
-                 Compiler (reserved=162KB, committed=162KB)
                           (malloc=63KB, #229)
                           (arena=99KB, #3)
```

类似地，这个区域是供编译器自身操作使用的，这与生成的代码放在代码缓存中是不同的。

```
-                   Symbol (reserved=12093KB, committed=12093KB)
                           (malloc=10039KB, #110773)
                           (arena=2054KB, #1)
```

保留字符串（Interned `String`）的引用与符号表引用放在这里。

```
-          Memory Tracking (reserved=22466KB, committed=22466KB)
                           (malloc=22466KB, #1872)
```

NMT 本身的操作也需要一些空间。

详细的内存跟踪信息

如果 JVM 是用 `-XX:NativeMemoryTracking=detail` 启动的，`jcmd`（最后加上一个 `detail` 参数）就会提供原生内存分配相关的非常详细的信息。其中会包括整个内存空间的一个映射，包括像这样的一些行：

```
0x00000006c0000000 - 0x00000007c0000000] reserved 4194304KB for Java Heap
        from [ReservedSpace::initialize(unsigned long, unsigned long,
                          bool, char*, unsigned long, bool)+0xc2]
```

```
                [0x00000006c0000000 - 0x00000006fb100000] committed 967680KB
                    from [PSVirtualSpace::expand_by(unsigned long)+0x53]
                [0x000000076ab00000 - 0x00000007c0000000] committed 1397760KB
                    from [PSVirtualSpace::expand_by(unsigned long)+0x53]
```

4 GB 的堆空间是在 initialize() 函数中保留的，两次分配实际是在 expand_by() 函数中进行的。

对于整个进程空间而言，这类信息是重复的。对于 JVM 工程师，它能提供很有意义的线索，但是对于我们这类开发人员，概要信息就够用了。

NMT 提供了两类关键信息：

总提交大小

进程的总提交大小，是该进程将要消耗的实际物理内存量。这个值和应用的 RSS（或工作集）很接近，但是操作系统提供的那些测量值存在一个问题，即有些内存虽然已经提交，但是其页面被置换出去了（paged out），RSS 是不会将其计算在内的。实际上，如果进程的 RSS 小于已提交内存，就通常表明操作系统很难将 JVM 的所有信息都放到物理内存中。

每部分的提交大小

当需要调优堆、代码缓存或元空间等不同部分的最大值时，了解此类内存在 JVM 中实际使用了多少非常有用。超量分配通常只会影响内存的保留，不过有些情况下，保留内存也很重要，而 NMT 可以帮助我们跟踪那些最大值可以再缩减的情况。

NMT跟踪

NMT 也支持跟踪内存分配随时间的变化情况。如果 JVM 在启动时启用了 NMT，可以使用如下命令确定内存的基线使用情况：

```
% jcmd process_id VM.native_memory baseline
```

这样，JVM 就会把当时的内存分配情况标记下来，作为基线。利用如下命令，可以比较 JVM 当前的内存分配情况与基线的差别：

```
% jcmd process_id VM.native_memory summary.diff
Native Memory Tracking:

Total:  reserved=5896078KB  -3655KB, committed=2358357KB -448047KB

-             Java Heap (reserved=4194304KB, committed=1920512KB -444927KB)
                        (mmap: reserved=4194304KB, committed=1920512KB -444927KB)
 ....
```

在这个例子中，JVM 保留了 5.8 GB 的内存，正在使用的是 2.3 GB。与基线相比，提交的内存减少了 448 MB。类似地，可以看出提交的堆内存减少了 444 MB（可以观察其余的输出内容，来确定另外 4 MB 内存是哪部分区域减少的）。

在检查 JVM 的内存占用随时间的变化情况时，这一技术非常有用。

快速小结

1. 在 Java 8 中，原生内存跟踪（NMT）提供了 JVM 所使用的原生内存的详细信息。从操作系统的角度看，其中包含 JVM 堆（对 OS 而言，堆也是原生内存的一部分）。
2. 对大多数分析而言，NMT 的概要模式足够了。它支持我们确定 JVM 提交了多少内存（以及这些内存用于干什么了）。

8.2 针对不同操作系统优化JVM

JVM 可以利用一些调优选项来优化操作系统内存的使用。

8.2.1 大页

一般用“页”这个术语来讨论内存分配和交换。页是操作系统管理物理内存的一个单元，还是操作系统分配内存的最小单元：要分配 1 个字节，操作系统一定会分配 1 个整页。程序中后续的内存分配都会从这个页获取，直到分配完毕，这时就会分配一个新页。

操作系统分配的页数一般要比物理内存能容纳的页数多很多，这就是存在分页机制的原因：地址空间中的页会被移入或移出交换空间（或其他存储，跟页中包含的内容有关）。这意味着，这些页和它们在计算机物理内存中所占的位置间存在某种映射。这些映射有两种不同的处理方式。所有的页映射都保存在一个全局页表中（操作系统可以扫描这个表，找到特定的映射），最常用的映射保存在 TLB（Translation Lookaside Buffers）中。TLB 保存在一个快速的缓存中，所以通过 TLB 表项访问页要比通过页表访问快得多。

机器中 TLB 表项的数目有限，TLB 会用作 LRU（Least Recently Used，最近最少使用的）缓存，因此最大化 TLB 表项的命中率就变得非常重要。因为每个表项表示一个内存页，所以增大应用所使用的页的大小一般会有所帮助。如果每个页能表示更多内存，则用更少的 TLB 表项就能涵盖整个程序的内存，这样当需要某个页时，在 TLB 中找到的可能性更大。一般而言，任何程序都是这样。具体到像 Java 应用服务器或堆为中等大小的其他 Java 程序，也是如此。

Java 支持 `-XX:+UseLargePages` 选项。其默认值跟具体的操作系统配置有关。在 Windows 上，必须在操作系统中启用大页。用 Windows 的术语来讲，这意味着支持各个用户锁定内存页（lock pages into memory），这在 Windows 的服务器版本中才能实现。在 Windows 操作系统上，除非显式启用了 `UseLargePages`，否则默认使用常规页。

在 Linux 上，`UseLargePages` 标志默认不会启用，要支持大页，也要配置一下操作系统。

在 Solaris 上，不需要什么操作系统方面的配置，默认启用大页。

如果在不支持大页的系统上启用了 `UseLargePages` 标志，JVM 不会给出警告，它会使用常规页。如果系统支持大页，但是没有大页可用（可能因为所有的大页都被用了，也可能也为操作系统配置错误），这时 JVM 会打印警告。

1. Linux大页

在 Linux 上，大页的配置会随发行版的不同而有所不同；想要获得最准确的说明，请参考所用发行版的文档。一般而言，可以分为如下 5 个步骤。

(1) 确定内核支持哪些大页大小。大页大小与计算机的处理器和内核启动参数有关，不过最常见的是 2 MB。

```
# grep Hugepagesize /proc/meminfo
Hugepagesize:       2048 kB
```

(2) 计算需要多少大页。如果 JVM 会分配 4 GB 大小的堆，系统支持 2 MB 的大页，则这个堆就需要 2048 个大页。系统可以使用的大页的数目是在 Linux 内核中全局定义的，因此要对将在该系统中运行的所有 JVM（以及其他任何会使用大页的程序）重复这个过程。考虑到非堆部分也有可能会使用大页，所以应多估算 10%（这样，这个例子要使用 2200 个大页）。

(3) 将这个值写到操作系统中（以便立即生效）：

```
# echo 2200 > /proc/sys/vm/nr_hugepages
```

(4) 将该值保存到 /etc/sysctl.conf 中，这样系统重启后这个值也会保存下来：

```
sys.nr_hugepages=2200
```

(5) 在很多 Linux 版本上，一个用户可以分配的内存页数量可能是有限的。编辑 /etc/security/limits.conf 文件，为运行 JVM 的用户（例如这个例子中的 `appuser`）添加 `memlock` 条目：

```
    appuser soft    memlock    4613734400
    appuser hard    memlock    4613734400
```

在修改了 limits.conf 文件之后，用户必须重新登录，这个值才会生效。重启之后，JVM 就应该能够分配必要的大页了。要验证其效果，运行如下命令：

```
# java -Xms4G -Xmx4G -XX:+UseLargePages -version
java version"1.7.0_17"
Java(TM) SE Runtime Environment (build 1.7.0_17-b02)
Java HotSpot(TM) 64-Bit Server VM (build 23.7-b01, mixed mode)
```

若这个命令成功完成，说明大页已经正确配置。如果大页内存配置不正确，则会出现如下警告：

```
Java HotSpot(TM) 64-Bit Server VM warning:
Failed to reserve shared memory (errno = 22).
```

2. Linux透明大页

Linux 内核从 2.6.32 版本开始支持透明大页，这种机制不再需要上述配置。不过仍然需要为 Java 开启透明大页，这可以通过修改 /sys/kernel/mm/transparent_hugepage/enabled 来实现：

```
# cat /sys/kernel/mm/transparent_hugepage/enabled
always [madvise] never
# echo always > /sys/kernel/mm/transparent_hugepage/enabled
# cat /sys/kernel/mm/transparent_hugepage/enabled
[always] madvise never
```

该文件中的默认值（见第一条命令的输出）是 `madvise`。只有明确告诉内核会使用大页的程序才能使用大页。因为无法通过 JVM 做到这一点，所以要将默认值设置为 `always`（通过第二条命令）。注意，这会影响该系统上的 JVM 和其他任何程序；它们运行的时候都会使用大页。

如果启用了透明大页，就不要指定 `UseLargePages` 标志。如果显式地设置了该标志，JVM 会使用传统的大页；如果没有配置好传统的大页，则使用标准页。如果该标志设置为默认值，则 JVM 会使用透明大页（如果已经配置）。

3. Windows大页

只有服务器版的 Windows 才支持大页。Windows 7 上的具体操作如下，不同版本间会有一些差别。

(1) 启动 Microsoft 管理控制台（Microsoft Management Center）。点击开始菜单，在搜索框中输入 `mmc`。
(2) 如果左侧的面板中没有出现本地计算机策略图标，则从“文件”菜单中选择“添加/删除管理单元”，添加“组策略对象编辑器”。如果找不到该选项，就说明当前使用的 Windows 版本不支持大页。
(3) 在左侧面板中，展开本地计算机策略→计算机配置→ Windows 配置→安全配置→本地策略，点击“用户权限分配”文件夹。
(4) 在右侧面板中，双击“锁定内存页”。
(5) 在弹出菜单中，添加用户或组。
(6) 点击确定。
(7) 退出 Microsoft 管理控制台。
(8) 重新启动。

重启之后，JVM 就应该能够分配必要的大页了。要验证其效果，运行如下命令：

```
# java -Xms4G -Xmx4G -XX:+UseLargePages -version
java version "1.7.0_17"
Java(TM) SE Runtime Environment (build 1.7.0_17-b02)
Java HotSpot(TM) 64-Bit Server VM (build 23.7-b01, mixed mode)
```

如果命令像上面这样成功完成，大页就设置正确了。如果配置不正确，会出现如下警告：

```
Java HotSpot(TM) Server VM warning: JVM cannot use large page memory
because it does not have enough privilege to lock pages in memory.
```

请记住，在不支持大页的 Windows 系统（比如“home”版本）上，这条命令不会打印错误：不管命令行设置的是什么，JVM 一旦发现操作系统不支持大页，就会将 `UseLargePages` 标志设置为 `false`。

4. 大页的大小

在大多数 Linux 和 Windows 系统上，操作系统会使用 2 MB 大小的大页，但这个数字会随操作系统的配置而变化。

严格来讲，处理器定义了可能的页大小。大部分当前的 Intel 和 SPARC 处理器支持很多可能的页大小：4 KB、8 KB、2 MB 和 256 MB，等等。然而，实际可以分配多大的页是由操作系统决定的。在 Solaris 上，处理器所支持的各种页大小均能支持，JVM 可以自由分配任意大小的页。在 Linux 内核上（至少在本书写作时），可以在内核启动时指定使用处理器所支持哪种页大小，不过这就是应用实际可以分配的唯一的大页大小。在 Windows 上，大页固定为 2 MB（同样，本书写作时是这样）。

为支持 Solaris，Java 支持通过 `-XX:LargePageSizeInBytes=N` 标志来设置要分配的大页大小。该标志默认值为 0，这意味着 JVM 应该选择特定于处理器的大页大小。

这个标志在各种平台上均可设置，而且 JVM 不会明示是否使用了指定的页大小。如果在某个 Linux 系统上分配一个非常大的堆，你可能认为应该设置 `-XX:LargePageSizeInBytes=256M`，以便 TLB 命中率达到最佳。可以这么做，而且 JVM 不会抱怨什么，但它仍然只会分配 2 MB 大小（或者是指定内核支持的某个页大小）的页。事实上，指定根本没有任何意义的页大小都是可能的，比如 `-XX:LargePageSizeInBytes=11111`。因为这个大小是不可用的，JVM 会直接选择该平台的默认页大小。

因此，至少就目前而言，这个标志实际只有在 Solaris 上才有用。在 Solaris 上，为了使用更大的页，可以选择比默认值（在 SPARC 处理器上默认是 4 MB）更大的页大小。在配备了大量内存的系统上，这会增加能够容纳进 TLB 缓存的页数，提高性能。要查看 Solaris 上可用的页大小，可以使用 `pagesize -a` 命令。

快速小结

1. 使用大页通常可以明显提升应用的速度。
2. 在大多数操作系统上，必须显式开启大页支持。

8.2.2 压缩的oop

第 4 章曾提到，对于同一任务，32 位 JVM 的性能要比 64 位 JVM 的好 5%~20%。当然，这要假定应用可以容纳进 32 位的进程空间中，这限定了堆要小于 4 GB。（在实践中，这往往意味着堆要小于 3.5 GB，因为 JVM 还需要一些原生内存空间；而且在某些 Windows 版本上，限制是 3 GB。）

性能差距是 64 位的对象引用导致的。主要原因是，在堆中，32 位的对象引用占 4 字节，而 64 位的对象引用占 8 字节，是前者的 2 倍。这就致使需要更多 GC 周期，因为堆中留给其他数据的空间少了。

JVM 可以使用压缩的 oop 来弥补额外的内存消耗。“oop”代表的是“ordinary object pointer”，即普通对象指针，JVM 将其用作对象引用的句柄。在 oop 只有 32 位长时，只能引用 4 GB 内存（2^{32}），这就是为什么 32 位 JVM 有 4 GB 堆空间限制的原因。（同样的限

制也适用于操作系统层面，32 位的进程的地址空间限制为 4 GB。）而在 oop 为 64 位长时，可以引用的内存就是 TB 级了。

有一个中间方案：如果有 35 位的 oop，又会怎么样呢？这样的指针可以引用 32 GB 的内存（2^{35}），在堆中占的空间也比 64 位的引用少。问题是没有 35 位长的寄存器可以存放这样的引用。不过 JVM 可以假设引用的最后 3 位都是 0。现在，就不是所有的引用都能保存在堆中了。当应用被存入 64 位的寄存器时，JVM 可以将其左移 3 位（末尾添加 3 个 0）。而当从寄存器读出时，JVM 又可以右移 3 位，丢弃末尾的 0。

这样 JVM 就有了可以引用 32 GB 内存的指针，而且每个指针在堆中只占用 32 位。然而这也意味着，对于不能被 8 整除的地址上的任何一个对象，JVM 都无法访问，因为从压缩的 oop 得到的任何地址均以 3 个 0 结尾。第一个可能的 oop 是 0x1，移位之后是 0x8。下一个 oop 是 0x2，在移位后变成了 0x10（16）。所以对象必须位于 8 字节的边界上。

其实在 JVM 中（不管是 32 位的还是 64 位的），对象已经按 8 字节边界对齐了；对于大部分处理器，这种对齐方案都是最优的。所以使用压缩的 oop 并不会损失什么。如果 JVM 中的第一个对象保存到位置 0，占用 57 字节，那下一个对象就要保存到位置 64，浪费了 7 字节，无法再分配。这种内存取舍是值得的（而且不管是否使用压缩的 oop，都是这样），因为在 8 字节对齐的位置，对象可以更快地访问。

不过这也是为什么 JVM 没有尝试模仿 36 位引用（可以访问 64 GB 的内存）的原因。在那种情况下，对象就要在 16 字节的边界上对齐，在堆中保存压缩指针所节约的成本，就被为对齐对象而浪费的内存抵消了。

这里有两点启示。第一，对于大小在 4 GB 和 32 GB 之间的堆，应该使用压缩的 oop。压缩的 oop 可以使用 `-XX:+UseCompressedOops` 标志启用；在 Java 7 和更新的版本中，只要堆的最大值小于 32 GB，压缩的 oop 默认就是启用的。（在 7.2.1 节我们曾指出，默认情况下，在堆空间为 32 GB 的 64 位 JVM 上，对象引用的大小为 4 个字节，这是因为压缩 oop 默认就是启用的。）

第二，使用了 31 GB 的堆，并启用压缩 oop 的程序，通常要快于使用了 33 GB 的堆的程序。尽管后者的堆更大，但是堆中的指针要占据额外的空间，这意味着更大的堆执行 GC 的频率会更频繁，性能也更差。

因此，最好是使用小于 32 GB 的堆空间，或者使用比 32 GB 至少多若干 GB 的堆空间。如果有额外的空间来弥补非压缩引用所使用的空间，GC 周期数就会有所减少。但是增加多少内存可以改善非压缩 oop 对 GC 的影响，并没有硬性的规则。不过平均而言，对象引用会占用 20% 的堆空间，所以 38 GB 是个不错的起点。

快速小结

1. 压缩的 oop 会在最有用的时候默认开启。
2. 使用了压缩 oop 的 31 GB 的堆，与稍微大一些、但因为堆太大而无法使用压缩 oop 的堆相比，性能通常要好一些。

8.3　小结

尽管 Java 的堆是最受关注的内存区域，但整个 JVM 的内存占用对性能至关重要，特别是与操作系统相关的部分。可以利用本章所探讨的工具跟踪 JVM 内存占用随时间的变化情况（关键在于关注 JVM 的提交内存，而非保留内存）。

也可以通过调整 JVM 使用操作系统内存（特别是大页）的方式来改进性能。对于长期运行的，特别是堆特别大的 JVM 而言，使用大页几乎总是有好处的。

第 9 章

线程与同步的性能

从刚问世起，Java 的部分魅力就来自其多线程。即便在多核和多 CPU 系统司空见惯之前，能够轻松编写多线程程序也是 Java 的一个标志性特征。

Java 性能方面的吸引力显而易见：如果有两个 CPU 可用，那么一个应用能够完成的工作量可能是原来的 2 倍，或者是以快 1 倍的时间完成相同的工作量。当然，这是在假设任务可以分解成离散的片段的前提之下的，因为 Java 不是能自动找出算法性部分并实现并行化的语言。幸运的是，今日所见之计算，往往是离散性的任务：服务器处理来自离散的客户端的同步请求，批处理作业在一系列数据上执行相同的操作，数学算法可以分节成多个组成部分，诸如此类。

本章探讨的主题是，如何挖掘出 Java 线程和同步设施的最大性能。

9.1 线程池与ThreadPoolExecutor

在 Java 中，线程可以使用定制的代码来管理；应用也可以利用线程池。Java EE 应用服务器就是围绕用一个或多个线程池处理请求这一概念构建的：对服务器上 Servlet 的每个调用都是通过池中的线程处理的（也有可能不同）。类似地，其他应用可以使用 Java 的 `ThreadPoolExecutor` 并行执行任务。

事实上，有些 Java EE 应用服务器就是使用 `ThreadPoolExecutor` 类的实例来管理其任务的，尽管有些应用服务器编写了自己的线程池，不过一般也仅仅是因为当时 Java API 中还没有加入 `ThreadPoolExecutor` 类而已。不过在这些情况下，线程池的实现可能会有所不同，但基本概念是一样的，本节都会予以讨论。

在使用线程池时，有一个因素非常关键：调节线程池的大小对获得最好的性能至关重要。线程池的性能会随线程池大小这一基本选择而有所不同，在某些条件下，线程池过大对性

能也有很大的不利影响。

所有线程池的工作方式本质是一样的：有一个队列，任务被提交到这个队列中。（可以有不止一个队列，概念是一样的。）一定数量的线程会从该队列中取任务，然后执行。任务的结果可以发回客户端（比如应用服务器的情况下），或保存到数据库中，或保存到某个内部数据结构中，等等。但是在执行完任务后，这个线程会返回任务队列，检索另一个任务并执行（如果没有更多任务要执行，该线程会等待下一个任务）。

线程池有最小线程数和最大线程数。池中会有最小数目的线程随时待命，等待任务指派给它们。因为创建线程的成本非常高昂，这样可以提高任务提交时的整体性能：已有的线程会拿到该任务并处理。另一方面，线程需要一些系统资源，包括栈所需的原生内存，如果空闲线程太多，就会消耗本来可以分配给其他进程的资源。最大线程数还是一个必要的限流阀，防止一次执行太多线程。

`ThreadPoolExecutor` 和相关的类将最小线程数称作核心池大小，大部分应用服务器会使用类似 minimum（最小值）的术语（如 `MinThreads`）。不要被术语所迷惑：它们是同一个概念。然而，在决定何时调整线程池大小的方式上，`ThreadPoolExecutor` 和大部分 Java EE 应用服务器有些重要的差别。本节后面会探讨这些差别。现在，考虑 `ThreadPoolExecutor` 的最简单的情况，大部分 Java EE 应用服务器也是这么工作的：如果有个任务要执行，而所有的并发线程都在忙于执行另一个任务，就启动一个新线程（直到创建的线程达到最大线程数）。

9.1.1　设置最大线程数

先来设定最大线程数：对于给定硬件上的给定负载，最大线程数设置为多少最好呢？这个问题回答起来并不简单；它取决于负载特性以及底层硬件。特别是，最优线程数还与每个任务阻塞的频率有关。

为方便讨论，假设 JVM 有 4 个 CPU 可用。至于是系统只有 4 个 CPU，还是说有 128 个硬件线程但我们只想利用其中的 4 个，并不重要，因为我们的目标就是最大化这 4 个 CPU 的利用率。

很明显，最大线程数至少要设置为 4。的确，除了处理这些任务，JVM 中还有些线程要做其他的事，但是它们几乎从来不会占用一个完整的 CPU。如果使用的是第 5 章所讨论的并发垃圾收集器，这是个例外，后台线程必须有足够的 CPU 来运行，以免在处理堆这方面落后。

如果线程数多于 4，会有帮助吗？这时就要看负载特性了。考虑最简单的情况，假定任务都是计算密集型的：没有外部网络调用（比如不会访问数据库），也不会激烈地竞争内部锁。在使用模实体管理器（mock entity manager）的情况下，股价历史批处理程序就是一个这样的应用：实体上的数据完全可以并行计算。

下面就使用线程池计算一下 10 000 个模股票实体的历史，假设机器有 4 个 CPU，使用不同的线程数测试，具体的性能数据见表 9-1。如果池中只有 1 个线程，计算数据集需要 255.6 秒；用 4 个线程，则只需要 77 秒。如果线程数超过 4 个，随着线程数的增加，需要的时间会稍多一些。

表9-1：计算10 000个模的价格历史所需时间

线程数	所需秒数	与基准的百分比
1	255.6	100%
2	134.8	52.7%
4	77.0	30.1%
8	81.7	31.9%
15	85.6	33.5%

如果应用中的任务是完全并行的，则在有 2 个线程时，“与基准的百分比”这列为 50%；在有 4 个线程时，这列为 25%。但是这种完全线性的比例不可能出现，原因有这么几点：如果没有其他线程帮助，这些线程必须自己来协同，实现从运行队列中选取任务（一般而言，通常会有更多同步）。到了使用 4 个线程的时候，系统会 100% 消耗可用的 CPU，尽管机器可能没有运行其他用户级的应用，但是会有各种系统级的进程进来，并使用 CPU，从而使得 JVM 无法 100% 地使用所有 CPU 周期。

尽管如此，这个应用在伸缩性方面表现还不错，且即使池中的线程数被显著高估，性能损失也比较轻微。

不过在其他情况下，性能损失可能会很大。在 Servlet 版的股票历史计算程序中，线程太多的话，影响会很大，如表 9-2 所示。应用服务器分别配置成不同的线程数，有一个负载生成器会向该服务器发送 20 个同步的（simultaneous）请求。

表9-2：每秒通过Servlet的操作

线程数	每秒操作数	与基准的百分比
4	77.43	100%
8	75.93	98.8%
16	71.65	92.5%
32	69.34	89.5%
64	60.44	78.1%

鉴于应用服务器有 4 个 CPU 可用，最大吞吐量可以通过将池中的线程数设置为 4 来实现。

第一章曾探讨过，在研究性能问题时确定瓶颈在哪儿比较重要。在这个例子中，瓶颈很明显是 CPU：4 个线程时，CPU 利用率为 100%。不过加入更多线程的影响其实很小，至少当线程数是原来的 8 倍时才会有明显的差别。

如果瓶颈在其他地方呢？这个例子有点不同寻常，任务完全是 CPU 密集型的：没有 I/O。一般来说，线程有可能会调用数据库，或者把输出写到某个地方，甚至是会合其他某些资源。在那种情况下，瓶颈未必是 CPU，而可能是外部资源。

对于此类情况，添加线程非常有害。虽然我在第一章只是半开玩笑地说过数据库总是瓶颈，但是瓶颈可能是任何外部资源。

仍以股票 Servlet 为例，我们把目标变一下：如果目标是最大限度地利用负载生成器机器，又会如何，是简单地运行一个多线程的 Java 程序吗？

在典型的用法中，如果 Servlet 应用运行在一个有 4 个 CPU 的应用服务器上，而且只有一个客户端请求数据，那么，应用服务器大约会 25% 忙碌，客户端机器几乎总是空闲的。如果负载增加到 4 个并发的客户端，则应用服务器会 100% 忙碌，客户端机器可能只有 20% 的忙碌。

只看客户端，很容易得出这样的结论：因为客户端 CPU 大量过剩，应该可以添加更多线程，改善其伸缩性。表 9-3 说明了这种假设何其错误：当客户端再加入一些线程时，性能会受到极大影响。

表9-3：计算模拟股票价格历史的平均响应时间

客户端线程数	平均响应时间（秒）	与基准的百分比
1	0.05	100%
2	0.05	100%
4	0.05	100%
6	0.076	152%
8	0.104	208%
16	0.212	424%
32	0.437	874%
64	0.909	1818%

在这个例子中，一旦应用服务器成为瓶颈（也就是说，线程数达到 4 个时），向服务器增加负载是非常有害的——即使只是在客户端加了几个线程。

这个例子看上去可能有点有意为之。如果服务器已经是 CPU 密集型的，谁还会加入更多线程呢？之所以使用这个例子，只是因为它容易理解，而且仅使用了 Java 程序。这意味着读者自己就可以运行，并理解它是如何工作的，而不必设置数据库连接、模式（Schema）等选项。

需要指出的是，对于还要向 CPU 密集型或 I/O 密集型的机器发送数据库请求的应用服务器而言，同样的原则也成立。你可能只关注应用服务器 CPU，看到小于 100% 就感觉不错；看到有多余的请求要处理，就假定增加应用服务器的线程数是个不错的主意。结果会让人大吃一惊，因为在那种情况下增加线程数，实际上会降低整体吞吐量（影响可能非常明显），就像前面那个只有 Java 程序的例子一样。

了解系统真正瓶颈之所在非常重要的另一个原因是：如果还向瓶颈处增加负载，性能会显著下降。相反，如果减少了当前瓶颈处的负载，性能可能会上升。

这也是设计自我调优的线程池非常困难的原因所在。线程池通常对挂起了多少工作有所了解，甚至有多少 CPU 可用也可以知道，但是它们通常看不到所在的整个环境的其他方面。因此，当有工作挂起时，增加线程（这是很多自我调优的线程池的一个核心特性，也是 `ThreadPoolExecutor` 的某些配置）往往是完全错误的。

遗憾的是，设置最大线程数更像是艺术而非科学，原因也在于此。在现实中，测试条件下自我调优的线程池会实现可能性能的 80%~90%；而且就算高估了所需线程数，也可能只有很小的损失。但是当设置线程数大小这方面出了问题时，系统可能会在很大程度上出现

问题。就此而言，充足的测试仍然非常关键。

9.1.2 设置最小线程数

一旦确定了线程池的最大线程数，就该确定所需的最小线程数了。大部分情况下，开发者会直截了当地将它们设置为同一个值。

将最小线程数设置为其他某个值（比如 1），出发点是防止系统创建太多线程，以节省系统资源。因为每个线程都需要一定量的内存，特别是线程的栈（本章后面会讨论）。根据第 2 章的一般原则之一，所设置的系统大小应该能够处理预期的最大吞吐量，而要达到最大吞吐量，系统将需要创建所有那些线程。如果系统做不到这一点，那选择一个最小线程数也没什么帮助：如果系统达到了这样的条件——需要按所设置的最大线程数启动所有线程，而又无法满足，系统将陷入困境。创建最终可能会需要的所有线程，并确保系统可以处理预期的最大负载，这样更好。

另一方面，指定一个最小线程数的负面影响相当小。如果第一次就有很多任务要执行，会有负面影响：这时线程池需要创建一个新线程。创建线程对性能不利，这也是为什么起初需要线程池的原因，不过这种一次性的成本在性能测试中很可能察觉不到（只要这个线程仍然在池中）。

在批处理应用中，线程是在创建线程池时分配（如果将最大线程数和最小线程数设置为同一个值，就会出现这种情况），还是按需分配，并不重要：执行应用所需的时间是一样的。在其他应用中，新线程可能会在预热阶段分配（分配线程的总时间还是一样的），对性能的影响可以忽略不计。即使线程创建发生在可以测量的周期内，只要此类操作有限，也很有可能测不出来。

另一个可以调优的地方是线程的空闲时间。比如，某个线程池的最小线程数为 1，最大线程数为 4。现在假设一般会有一个线程在执行，处理一个任务；然后应用进入这样一个循环：每 15 秒，负载平均有 2 个任务要执行。第一次进入这个循环时，线程池会创建第 2 个线程，此时，让这个新创建的线程在池中至少留存一段时间是有意义的。我们希望避免这种情况：第 2 个线程创建出来后，5 秒钟内结束其任务，空闲 5 秒，然后退出了。而 5 秒之后又需要为下一个任务创建一个线程。一般而言，对于线程数为最小值的线程池，一个新线程一旦创建出来，至少应该留存几分钟，以处理任何负载飙升。如果任务到达率有个比较好的模型，可以基于这个模型设置空闲时间。另外，空闲时间应该以分钟计，而且至少在 10 分钟到 30 分钟之间。

留存一些空闲线程，对应用性能的影响通常微乎其微。一般而言，线程对象本身不会占用大量的堆空间。除非线程保持了大量的线程局部存储，或者线程的 `Runnable` 对象引用了大量内存。不管是哪种情况，释放这样的线程都会显著减少堆中的活数据（这反过来又会影响 GC 的效率）。

不过对线程池而言，这些情况并不多见。当池中的某个对象空闲时，它就不应该再引用任何 `Runnable` 对象（如果引用了，就说明哪个地方有 bug 了）。根据线程池的实现情况，线程局部变量可能会继续保留；尽管在某些情况下，线程局部变量可以有效促成对象重用（参见第 7 章），但是那些线程局部对象所占用的总的内存量，应该加以限制。

对于可能会增长到非常大（当然也是运行在规模很大的机器上）的线程池，这个规则有个重要的特例。举例而言，假设某个线程池的任务队列预计平均有 20 个任务，那么 20 就是很好的最小值。再假设这个池运行在一个规模很大的机器上，它被设计为可以处理 2000 个任务的峰值负载。如果在池中留存 2000 个空闲线程，则当只有 20 个任务时，对性能会有所影响：如果只有核心的 20 个线程忙碌，与有 1980 个空闲线程相比，前者的吞吐量可能是后者的 50%。线程池一般不会遇到这样的问题，但如果遇到了，那就应该确认一下池的合适的最小值了。

9.1.3　线程池任务大小

等待线程池来执行的任务会被保存到某类队列或列表中；当池中有线程可以执行任务时，就从队列中拉出一个。这会导致不均衡：队列中任务的数量有可能变得非常大。如果队列太大，其中的任务就必须等待很长时间，直到前面的任务执行完毕。例如一个超负荷的 Web 服务器：如果有个任务被添加到队列中，但是没有在 3 秒钟内执行，那用户很可能就去看另一个页面了。

因此，对于容纳等待执行任务的队列，线程池通常会限制其大小。根据用于容纳等待执行任务的数据结构的不同，`ThreadPoolExecutor` 会有不同的处理方式（下一节会更详细地介绍）；应用服务器通常有一些调优参数，可以调整这个值。

就像线程池的最大线程数，这个值应该如何调优，并没有一个通用的规则。举例而言，假设某个应用服务器的任务队列中有 30 000 个任务，有 4 个 CPU 可用，如果执行一个任务只需要 50 毫秒，同时假设这段时间不会到达新任务，则清空任务队列需要 6 分钟。这可能是可以接受的，但如果每个任务需要 1 秒钟，则清空任务队列需要 2 小时。因此，若要确定使用哪个值能带来我们需要的性能，测量我们的真实应用是唯一的途径。

不管是哪种情况，如果达到了队列数限制，再添加任务就会失败。`ThreadPoolExecutor` 有一个 `rejectedExecution` 方法，用于处理这种情况（默认会抛出 `RejectedExecutionException`）。应用服务器会向用户返回某个错误：或者是 HTTP 状态码 500（内部错误），或者是 Web 服务器捕获错误，并向用户给出合理的解释消息——其中后者是最理想的。

9.1.4　设置ThreadPoolExecutor的大小

线程池的一般行为是这样的：创建时准备好最小数目的线程，如果来了一个任务，而此时所有的线程都在忙碌，则启动一个新线程（一直到达到最大线程数），任务就可以立即执行了。否则，任务被加入等待队列，如果任务队列中已经无法加入新任务，则拒绝之。不过，`ThreadPoolExecutor` 的表现可能和这种标准行为有点不同。

根据所选任务队列的类型，`ThreadPoolExecutor` 会决定何时启动一个新线程。有以下 3 种可能。

`SynchronousQueue`

　如果 `ThreadPoolExecutor` 搭配的是 `SynchronousQueue`，则线程池的行为会和我们预计的一样，它会考虑线程数：如果所有的线程都在忙碌，而且池中的线程数尚未达到最大，

则新任务会启动一个新线程。然而，这个队列没办法保存等待的任务：如果来了一个任务，创建的线程数已经达到最大值，而且所有线程都在忙碌，则新的任务总是会被拒绝。所以如果只是管理少量的任务，这是个不错的选择；但是对于其他情况，就不合适了。该类文档建议将最大线程数指定为一个非常大的值，如果任务完全是 CPU 密集型的，这可能行得通，但是我们会看到，其他情况下可能会适得其反。另一方面，如果需要一个容易调整线程数的线程池，这种选择会更好。

无界队列

如果 `ThreadPoolExecutor` 搭配的是无界队列（比如 `LinkedBlockedingQueue`），则不会拒绝任何任务（因为队列大小没有限制）。这种情况下，`ThreadPoolExecutor` 最多仅会按最小线程数创建线程，也就是说，最大线程池大小被忽略了。如果最大线程数和最小线程数相同，则这种选择和配置了固定线程数的传统线程池运行机制最为接近。

有界队列

在决定何时启动一个新线程时，使用了有界队列（如 `ArrayBlockingQueue`）的 `ThreadPoolExecutor` 会采用一个非常复杂的算法。比如，假设池的核心大小为 4，最大为 8，所用的 `ArrayBlockingQueue` 最大为 10。随着任务到达并被放到队列中，线程池中最多会运行 4 个线程（也就是核心大小）。即使队列完全填满，也就是说有 10 个处于等待状态的任务，`ThreadPoolExecutor` 也是只利用 4 个线程。

如果队列已满，而又有新任务加进来，此时才会启动一个新线程。这里不会因为队列已满而拒绝该任务，相反，会启动一个新线程。新线程会运行队列中的第一个任务，为新来的任务腾出空间。

在这个例子中，池中会有 8 个线程（最大线程数）的唯一一种情形是，有 7 个任务正在处理，队列中有 10 个任务，这时又来了一个新任务。

这个算法背后的理念是，该池大部分时间仅使用核心线程（4 个），即使有适量的任务在队列中等待运行。这时线程池就可以用作节流阀（这是很有好处的）。如果积压的请求变得非常多，该池就会尝试运行更多线程来清理；这时第二个节流阀——最大线程数——就起作用了。

如果系统没有外部瓶颈，CPU 周期也足够，那一切就都解决了：加入新的线程可以更快地处理任务队列，并很可能使其回到预期大小。该算法所适合的用例当然也很容易构造。

另一方面，该算法并不知道队列为何会突然增大。如果是因为外部的任务积压，那么加入更多线程并非明智之举。如果该线程所运行的机器已经是 CPU 密集型的，加入更多线程也是错误的。只有当任务积压是由额外的负载进入系统（比如有更多客户端发起 HTTP 请求）引发时，增加线程才是有意义的。（如果是这种情况，为什么要等到队列已经接近某个边界时才增加呢？如果有额外的资源供更多线程使用，则尽早增加线程将改善系统的整体性能。）

对于上面提到的每一种选择，都能找到很多支持或反对的论据，但是在尝试获得最好的性能时，可以应用 KISS 原则“Keep it simple, stupid”。 可以将 `ThreadPoolExecutor` 的核心线程数和最大线程数设为相同，在保存等待任务方面，如果适合使用无界任务列表，则选择 `LinkedBlockingQueue`；如果适合使用有界任务列表，则选择 `ArrayBlockingQueue`。

快速小结

1. 有时对象池也是不错的选择，线程池就是情形之一：线程初始化的成本很高，线程池使得系统上的线程数容易控制。
2. 线程池必须仔细调优。盲目向池中添加新线程，在某些情况下对性能会有不利影响。
3. 在使用 `ThreadPoolExecutor` 时，选择更简单的选项通常会带来最好的、最能预见的性能。

9.2 ForkJoinPool

Java 7 引入了一个新的线程池：`ForkJoinPool` 类。这个类看上去和其他任何线程池都很像；和 `ThreadPoolExecutor` 类一样，它也实现了 `Executor` 和 `ExecutorService` 接口。在支持这些接口方面，`ForkJoinPool` 在内部会使用一个无界任务列表，供构造器中所指定数目（如果所选的是无参构造器，则为该机器上的 CPU 数）的线程来运行。

`ForkJoinPool` 类是为配合分治算法的使用而设计的：任务可以递归地分解为子集。这些子集可以并行处理，然后每个子集的结果被归并到一个结果中。一个经典的例子就是快速排序算法。

分治算法的重点是，算法会创建大量的任务，而这些任务只有相对较少的几个线程来管理。比如要排序一个包含 1000 万个元素的数组。首先创建单独的任务来执行 3 个操作：排序包含前面 500 万个元素的子数组，再排序包含后面 500 万个元素的子数组，然后合并两个子数组。

类似地，要排序包含 500 万个元素的数组，可以分别排序包含 250 万个元素的子数组，然后合并子数组。一直递归到某个点（比如到子数组包含 10 个元素时），这时在子数组上使用插入排序直接处理更为高效。图 9-1 演示了其工作方式。

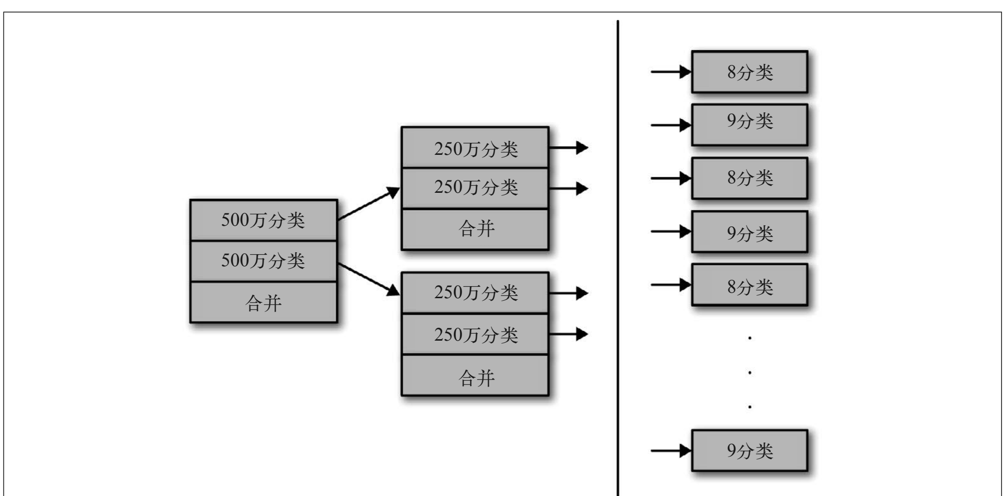


图 9-1：递归快速排序中的任务

最后会有超过 100 万个任务来排序叶子数组（每个数组少于 10 个元素，这时候直接排序即可；这里只是用 10 来举例，实际值会随实现的不同而有所变化。在目前的 Java 库实现中，当数组少于 47 个元素时[1]，会采用插入排序）。需要 50 多万个任务来归并那些排好序的数组，归并下一级又需要 25 万个任务，依此类推。最后会有 2 097 151 个任务。

更大的问题是，所有任务都要等待它们派生出的任务先完成，然后才能完成。对于元素数少于 10 的子数组，直接对它们做排序的任务必须优先完成；在此之后，创建相应子数组的任务才能归并其子数组的结果，依此类推：链条上的所有任务依次归并，直到整个数组被归并为最终的、排序好的结果。

因为父任务必须等待子任务完成，所以无法使用 ThreadPoolExecutor 高效实现这个算法。ThreadPoolExecutor 内的线程无法将另一个任务添加到队列中并等待其完成：一旦线程进入等待状态，就无法使用该线程执行它的某个子任务了。另一方面，ForkJoinPool 则允许其中的线程创建新任务，之后挂起当前的任务。当任务被挂起时，线程可以执行其他等待的任务。

举个简单的例子：比如说有个 double 数组，我们想计算数组中小于 0.5 的元素的个数。顺序扫描比较简单（可能还有优势，本节后面会看到），但是为了说明问题，现在把数组划分为子数组，并行扫描（模仿更复杂的快速排序和其他分治算法）。使用 ForkJoinPool 实现这一功能的代码如下：

```
public class ForkJoinTest {
    private double[] d;

    private class ForkJoinTask extends RecursiveTask<Integer> {
        private int first;
        private int last;

        public ForkJoinTask(int first, int last) {
            this.first = first;
            this.last = last;
        }

        protected Integer compute() {
            int subCount;
            if (last - first < 10) {
                subCount = 0;
                for (int i = first; i <= last; i++) {
                    if (d[i] < 0.5)
                        subCount++;
                }
            }
            else {
                int mid = (first + last) >>> 1;
                ForkJoinTask left = new ForkJoinTask(first, mid);
                left.fork();
                ForkJoinTask right = new ForkJoinTask(mid + 1, last);
                right.fork();
```

注 1：可以参考 java.util.DualPivotQuicksort 的实现。——译者注

```
                subCount = left.join();
                subCount += right.join();
            }
            return subCount;
        }
    }

    public static void main(String[] args) {
        d = createArrayOfRandomDoubles();
        int n = new ForkJoinPool().invoke(new ForkJoinTask(0, 9999999));
        System.out.println("Found " + n + " values");
    }
}
```

`fork()` 和 `join()` 方法是这里的关键：没有这些方法，实现这类递归会非常痛苦（在由 `ThreadPoolExecutor` 执行的任务中就没有这些方法）。这些方法使用了一系列内部的、从属于每个线程的队列来操纵任务，并将线程从执行一个任务切换到执行另一个。细节对开发者是透明的，不过如果对算法感兴趣，其代码读起来也很有意思。这里我们重点关注的是性能：`ForkJoinPool` 和 `ThreadPoolExecutor` 这两个类之间有什么权衡取舍呢？

首先，fork/join 范型所实现的挂起，使得所有任务可以交由少量的线程执行。使用该示例代码计算包含 1000 万个元素的数组中的 `double` 值，会创建 200 多万个任务，但这些任务很容易交由少量一些线程执行（甚至是一个线程，如果这对运行测试的机器有意义的话）。使用 `ThreadPoolExecutor` 运行类似算法则需要 200 多万个线程，因为每个线程必须等待其子任务完成，而且那些子任务只有在池中有可用线程时才能完成。有了 fork/join，我们可以实现用 `ThreadPoolExecutor` 无法实现的算法，这就是一个性能优势。

尽管分治技术非常强大，但是滥用也可能会导致性能变糟糕。在计数的这个例子中，可以使用一个线程来扫描数组并计数，虽然未必能像并行运行 fork/join 算法那样快。然而，把原数组划分为多个断，使用 `ThreadPoolExecutor` 让多个线程扫描数组，也是非常容易的：

```
public class ThreadPoolTest {
    private double[] d;

    private class ThreadPoolExecutorTask implements Callable<Integer> {
        private int first;
        private int last;

        public ThreadPoolExecutorTask(int first, int last) {
            this.first = first;
            this.last = last;
        }

        public Integer call() {
            int subCount = 0;
            for (int i = first; i <= last; i++) {
                if (d[i] < 0.5) {
                    subCount++;
                }
            }
            return subCount;
        }
```

```
    }

    public static void main(String[] args) {
        d = createArrayOfRandomDoubles();
        ThreadPoolExecutor tpe = new ThreadPoolExecutor(4, 4,
                                        Long.MAX_VALUE,
                                        TimeUnit.SECONDS,
                                        new LinkedBlockingQueue());
        Future[] f = new Future[4];
        int size = d.length / 4;
        for (int i = 0; i < 3; i++) {
            f[i] = tpe.submit(
                    new ThreadPoolExecutorTask(i * size, (i + 1) * size - 1);
        }
        f[3] = tpe.submit(new ThreadPoolExecutorTask(3 * size, d.length - 1);
        int n = 0;
        for (int i = 0; i < 4; i++) {
            n += f.get();
        }
        System.out.println("Found " + n + " values");
    }
}
```

在一个配备了 4 个 CPU 的机器上，这段代码可以充分利用所有可用的 CPU，并行处理数组，同时避免像 fork/join 示例中那样创建和排队处理 200 万个任务。可以预见性能会快些，如表 9-4 所示。

表9-4：对1亿个元素做计数处理

线程数	ForkJoinPool（秒）	ThreadPoolExecutor（秒）
1	3.2	0.31
4	1.9	0.15

测试所用的机器有 4 个 CPU，4 GB 固定内存。测试中，ThreadPoolExecutor 完全不需要 GC，而每个 ForkJoinPool 测试会花 1.2 秒在 GC 上。对于性能差异而言，这一点所占比重很大，但这并非故事的全部：创建和管理任务对象的开销也会伤害 ForkJoinPool 的性能。如果有类似的替代方案，很可能会更快，至少在这个简单的例子中是这样。

ForkJoinPool 还有一个额外的特性，它实现了工作窃取（work-stealing）。这基本上就是一个实现细节了；这意味着池中的每个线程都有自己所创建任务的队列。线程会优先处理自己队列中的任务，但如果这个队列已空，它会从其他线程的队列中窃取任务。其结果是，即使 200 万个任务中有一个需要很长的执行时间，ForkJoinPool 中的其他线程也可以完成其余的随便什么任务。ThreadPoolExecutor 则不会这样：如果一个任务需要很长的时间，其他线程并不能处理额外的工作。

示例代码先是计算数组中小于 0.5 的元素数。此外，如果代码中还计算了一个新的值，并保存到数组中了，会发生什么？一个没有实际意义但却是 CPU 密集型的实现可以执行以下代码：

```
for (int i = first; i <= last; i++) {
    if (d[i] < 0.5) {
```

```
            subCount++;
        }
        for (int j = 0; j < d.length - i; j++) {
            for (int k = 0; k < 100; k++) {
                dummy = j * k + i; // dummy is volatile, so multiple writes occur
                d[i] = dummy;
            }
        }
    }
```

因为用 j 索引的外部循环是基于元素在数组中的位置处理的，所以计算所需要的时间和元素位置成比例关系：计算 d[0] 的值需要很长的时间，而计算 d[d.length - 1] 则只需要很短的时间。

简单地将数组分为 4 段，用 ThreadPoolExecutor 处理，这个测试有一个不好的地方。计算数组第 1 段的线程需要很长的时间才能完成，比处理数组最后一段的第 4 个线程所需的时间长得多。一旦第 4 个线程结束，它就会处于空闲状态：所有线程都要等第 1 个线程完成它的耗时较长的任务。

在粒度为 200 万个任务的 ForkJoinPool 中，尽管有一个线程会忙于针对数组中的前 10 个元素的非常耗时的计算，但是其余线程都有工作可做，在大部分测试过程中，CPU 会保持忙碌。区别如表 9-5 所示。

表9-5：处理包含10 000个元素的数组的时间

线程数	ForkJoinPool（秒）	ThreadPoolExecutor（秒）
1	54.5	53.3
4	16.6	24.2

当池中只有一个线程时，计算所花的时间基本一样。这可以理解：不管池如何实现，计算量是一样的；而且因为那些计算绝对不会并行进行，所以可以预计它们所需的时间是一样的（尽管创建 200 万个任务会有少量开销）。但是当池中包含 4 个线程时，ForkJoinPool 中任务的粒度会带来一个决定性的优势：几乎在测试的整个过程中，都能保持 CPU 的忙碌状态。

这种情况就叫作“不均衡”，因为某些任务所花的时间比其他任务长（因此前面例子中的任务可以说是“均衡的”）。一般而言，如果任务是均衡的，使用分段的 ThreadPoolExecutor 性能更好；而如果任务是不均衡的，则使用 ForkJoinPool 性能更好。

还有一个更微妙的性能方面的建议：请仔细考虑 fork/join 范型应该在哪个点结束递归。在这个例子中，我信手选择了当数组大小小于 10 时结束。如果在数组大小为 250 万时停止递归，那么 fork/join 测试（在搭载 4 个 CPU 的机器上，处理 1000 万个元素的平衡代码）会只创建 4 个任务，其性能基本和 ThreadPoolExecutor 一样。

另一方面，对于这个例子，在非平衡的测试中，继续递归会有更好的性能，即使创建更多任务。表 9-6 给出了一些有代表性的数据点。

表9-6：处理包含10 000个元素的数组的时间

叶子数组大小	ForkJoinPool（秒）
20	17.8
10	16.6
5	15.6
1	16.8

自动并行化

Java 8 向 Java 中引入了自动并行化特定种类代码的能力。这种并行化就依赖于 ForkJoinPool 类的使用。Java 8 为这个类加入了一个新特性：一个公共的池，可供任何没有显式指定给某个特定池的 ForkJoinTask 使用。这个公共池是 ForkJoinPool 类的一个 static 元素，其大小默认设置为目标机器上的处理器数。

这种并行化在 Arrays 类的很多新方法中都会发生，包括使用并行快速排序处理数组的方法，操作数组的每个元素的方法，等等。在 Java 8 的 Stream 特性中也有应用，支持在集合中的每个元素上（或顺序或并行地）执行操作。Stream 的一些基本的性能特性会在第 12 章讨论；在本节中，我们看一下 Stream 是如何自动地并行处理的。

给定一个包含一系列整型数的集合，下列代码会计算与给定整型数匹配的股票代号的价格历史：

```
Stream<Integer> stream = arrayList.parallelStream();
stream.forEach(a -> {
    String symbol = StockPriceUtils.makeSymbol(a);
    StockPriceHistory sph = new StockPriceHistoryImpl(symbol, startDate,
                                  endDate, entityManager);
});
```

这段代码会并行计算模拟价格历史：forEach() 方法将为数组列表中的每个元素创建一个任务，每个任务都会由公共的 ForkJoinTask 池处理。它在功能上与本章开始所做的测试是等价的，那个测试是用一个线程池来并行计算价格历史（不过与显式使用线程池相比，这段代码写起来更容易）。

设置 ForkJoinTask 池的大小和设置其他任何线程池同样重要。默认情况下，公共池的线程数等于机器上的 CPU 数。如果在同一机器上运行着多个 JVM，则应限制这个线程数，以防这些 JVM 彼此争用 CPU。类似地，如果 Servlet 代码会执行某个并行任务，而我们想确保 CPU 可供其他任务使用，可以考虑减小公共池的线程数。另外，如果公共池中的任务会阻塞等待 I/O 或其他数据，也可以考虑增大线程数。

这个值可以通过设置系统属性 -Djava.util.concurrent.ForkJoinPool.common.parallelism=*N* 来指定。

在本章前面的表 9-1 中，曾经对比过线程数对并行计算股票历史价格的影响。表 9-7 使用共同的 ForkJoinPool（将 parallelism 系统属性设置为给定的值）将那个数据与 forEach() 构造作了比较。

表9-7：计算10 000支模拟股票价格历史所需的时间

线程数	ThreadPoolExecutor（秒）	ForkJoinPool（秒）
1	255.6	135.4
2	134.8	110.2
4	77.0	96.5
8	81.7	84.0
15	85.6	84.6

默认情况下，公共池有 4 个线程（在这个配置了 4 个 CPU 的机器上），所以表中的第 3 行为一般情况。在线程数为 1 和 2 时，这类结果会让性能工程师很不开心：它们看上去很不协调，而当某一项测试出现这样的情况时，最常见的原因是测试错误。这里的原因是 `forEach()` 方法有些奇怪的行为：它使用了一个线程执行语句，还使用了公共池中的线程处理来自 Stream 的数据。即使在第 1 个测试中，公共池也是配置为使用一个线程，总的还是会使用两个线程来计算结果。（因此，使用了 2 个线程的 `ThreadPoolExecutor` 和使用了 1 个线程的 `ForkJoinPool` 的耗时基本相同。）

在使用并行 Stream 构造或其他自动并行化特性时，如果需要调整公共池的大小，可以考虑将所需的值减 1。

快速小结

1. `ForkJoinPool` 类应该用于递归、分治算法。
2. 应该花些心思来确定，算法中的递归任务何时结束最为合适。创建太多任务会降低性能，但如果任务太少，而任务所需的执行时间又长短不一，也会降低性能。
3. Java 8 中使用了自动并行化的特性会用到一个公共的 `ForkJoinPool` 实例。我们可能需要根据实际情况调整这个实例的默认大小。

9.3 线程同步

在理想的世界中，或者是在书本上的例子中，很容易避开对线程同步的需求。而在现实世界中，就未必那么容易了。

同步与 Java 并发设施

在本节中，当用到“同步”（synchronization）这个术语时，它指的是这样的代码：这段代码在一个代码块内，它们对一组变量的访问看上去是串行的，每次只有一个线程能访问内存。具体而言，既包括用 `synchronized` 关键字保护的代码块，也包括用 `java.util.concurrent.lock.Lock` 实例保护的代码，再就是 `java.util.concurrent` 和 `java.util.concurrent.atomic` 包内的代码。

严格来讲，`atomic` 下的类并没有使用同步，至少从 CPU 编程术语来看是这样的。它们利用了“比较与交换”（Compare and Swap，CAS）CPU 指令，而同步需要互斥访问某

个资源。在同步访问同一资源时，利用了 CAS 指令的线程不会阻塞，而对于需要同步锁的线程而言，如果另一个线程占据了该资源，则这个线程会阻塞。

这两种方式之间存在性能的权衡（本节后面会讨论）。然而，即使 CAS 指令是无锁、非阻塞的，它们仍然会表现出阻塞方式所具有的大部分行为：在开发者看来，最终结果看上去还是线程只能串行地访问被保护内存。

9.3.1 同步的代价

同步代码对性能有两个方面的影响。其一，应用在同步块上所花的时间会影响该应用的可伸缩性。其二，获取同步锁需要一些 CPU 周期，所以也会影响性能。

1. 同步与可伸缩性

先看重要的，当某个应用被分割到多个线程上运行时，加速比（Speedup）可以用如下等式定义（即 Amdahl 定律）：

$$Speedup = \frac{1}{(1 - P) + \frac{P}{N}}$$

P 是程序并行运行部分所花的时间，N 是所用到的线程数（假定每个线程总有 CPU 可用）。所以，如果 20% 的代码是串行执行的（这意味着 P 是 80%），有 8 个 CPU 可用，则可以预计存在并发的情况下加速比为 3.33。

从这个等式可以看出一个关键事实，即随着 P 值的降低（也就是说，有更多代码是串行执行的），引入多个线程所带来的性能优势也会随之下降。限制串行块中的代码量之所以如此重要，原因就在于此。在这个例子中，有 8 个 CPU 可用，我们可能会希望速度提升 8 倍。但是在只有 20% 的代码串行执行时，引入多个线程的好处就少了一半多（只增加了 3.33 倍）。

2. 锁定对象的开销

除了对可伸缩性的影响，同步操作本身还有两个基本的开销。

首先是获取同步锁的成本。如果某个锁没有被争用（即两个线程没有同时尝试访问这个锁），那这方面的开销会相当小。`synchronized` 关键字和 CAS 指令之间有轻微的差别。非竞争的 `synchronized` 锁被称为非膨胀（uninflated）锁，获取非膨胀锁的开销在几百纳秒的数量级。非竞争的 CAS 代码损失会更小。（第 12 章有例子对比了其差别，可以参考。）

在存在竞争的条件下，开销会更高。当第 2 个线程尝试访问某个同步锁时，可以预见这个锁会变成膨胀的（inflated）。这个成本是固定的，不管是 2 个还是 20 个线程要访问同一个锁，执行的代码量是一样的。（20 个线程都必须执行加锁代码，当然，成本会随线程数增加，但每个线程所花的时间是固定的，这是重点。）

对于使用 CAS 指令的代码，当存在竞争时，开销是无法预测的。CAS 原语基于一种乐观的策略：线程设置某个值，执行一些代码，然后确保初始值没有被修改。如果值被修改了，那么基于 CAS 的代码必须再次执行这些代码。在最坏的情况下，如果有两个线程，

它们都在修改 CAS 所保护的值，那么相互就会看到另一个线程同时也在修改这个值，就有可能会陷入无限循环。不过在实践中，两个线程不会进入这样的无限循环，但是随着竞争 CAS 所保护值的线程数的增加，重试次数也会增加。（如果此处的操作是只读的，那基于 CAS 的保护不会受竞争访问的影响。比如，不管有多少线程，它们都可以同时在同一个对象上调用 `AtomicLong.get()` 方法，而不用因竞争付出任何代价。这是使用基于 CAS 的设施的另一个重要优势。）

同步的第 2 个开销是 Java 特有的，依赖于 Java 内存模型（Java Memory Model）。和 C++ 和 C 等语言不同，Java 对同步相关的内存语义有严格的保证，而且该保证适用于基于 CAS 的保护、传统的同步以及 `volatile` 关键字。

例子中使用的 volatile

Java 内存模型对本书中的两个例子有微妙的影响。第 2 章探讨了编写一个微基准测试存在的问题；最终的解决方案需要一个 `volatile` 变量来保存每次循环迭代的结果：

```
public class MicroBenchmark {
    private volatile double answer;
    public static void main(String[] args) {
        long then = System.currentTimeMills();
        for (int i = 0; i < nLoops; i++) {
            answer = compute(randomValue[i]);
        }
        long now = System.currentTimeMills();
        System.out.println("Elapsed time:" + (now - then));
    }
}
```

编译器会优化代码，它可以展开循环，得到如下伪代码：

```
for (int i = 0; i < nLoops; i += 4) {
    answer = compute(randomValue[i]);
    answer = compute(randomValue[i + 1]);
    answer = compute(randomValue[i + 2]);
    answer = compute(randomValue[i + 3]);
}
```

如果 JVM 把 `answer` 的值保存在某个寄存器中，它就可以注意到寄存器被写了多次，但是没有读取操作（因为其他线程不能读这个寄存器），因此，除了最终结果，可以优化掉所有的循环计算。将 `answer` 用 `volatile` 定义，确保 JVM 必须保存每次循环的计算结果。JVM 不能优化掉这些计算，因为它无从知道是否会有其他线程过来，从主内存（main memory）中读取这个值。

类似地，第 7 章中的双重检查锁定的例子中也需要使用一个 `volatile` 变量：

```
private volatile ConcurrentHashMap instanceChm;
...
public void doOperation() {
    ConcurrentHashMap chm = instanceChm;
    if (chm == null) {
```

```
            synchronized(this) {
                chm = instanceChm;
                if (chm == null) {
                    chm = new ConcurrentHashMap();
                    …… 填充这个Map
                    instanceChm = chm;
                }
            }
        }
        ……use the chm……
    }
```

在这个例子中，volatile 关键字实现了两个目的。其一，可以注意到，HashMap 是先用一个局部变量初始化的，而且只有完全初始化的最终值才会被赋值给 instanceChm 变量。如果填充 HashMap 的代码直接使用 instanceChm 这个实例变量，则第 2 个线程有可能会看到一个部分初始化的 Map。其二，它可以确保当 Map 完全初始化后，其他线程可以立即看到值被写入了 instanceChm 变量。

同步的目的是保护对内存中值（也就是变量）的访问。如第 4 章所讨论的，变量可能会临时保存在寄存器中，这要比直接在主内存中访问更高效。寄存器值对其他线程是不可见的；当前线程修改了寄存器中的某个值，必须在某个时机把寄存器中的值刷新到主内存中，以便其他线程可以看到这个值。而寄存器值必须刷新的时机，就是由线程同步控制的。

实际的语言会非常复杂，最简单的理解是，当一个线程离开某个同步块时，必须将任何修改过的值刷新到主内存中。这意味着进入该同步块的其他线程将能看到最新修改的值。类似地，基于 CAS 的保护确保操作期间修改的变量被刷新到主内存中，标记为 volatile 的变量，无论什么时候被修改了，总会在主内存中更新。

第 1 章曾提到，应该学习避免使用 Java 中性能不太高的构造，即使这看上去像“过早地优化”自己的代码（事实并非如此）。下面循环中有个有趣的案例，而且是一个现实中的例子：

```
Vector v;
for (int i = 0; i < v.size(); i++) {
    process(v.get(i));
}
```

在生产中，我们发现这个循环消耗的时间惊人。比较合乎逻辑的假设是，process() 方法是罪魁祸首。但事实并非如此，问题也不在于 size() 和 get() 方法调用本身（调用已经被编译器内联了）。Vector 类的 size() 和 get() 方法是同步的，所有这些调用所需要的寄存器刷新是很大的性能问题。

这段代码之所以不理想，还有其他一些原因。特别是，在某个线程调用 size() 和调用 get() 的中间时间内，Vector 对象的状态有可能会发生变化。如果在此之间，另一个线程移除了这个对象的最后一个元素，则 get() 方法将抛出 ArrayIndexOutOfBoundsException。除了代码中的语义问题，细粒度的同步也是较差的选择。

一种方案是将大量连续的、细粒度的同步调用包含在一个同步块内：

```
synchronized(v) {
    for (int i = 0; i < v.size(); i++) {
        process(v.get(i));
    }
}
```

如果 process() 方法执行时间很长，并不能很好地解决问题，因为这个 Vector 对象没法并行处理。另一个备选方案，复制并分割 Vector 对象可能是必要的，这样就可以借助副本来并行处理其中的元素，不过其他线程仍然可能会修改原始的 Vector 对象。

寄存器刷新的影响也和程序运行所在的处理器种类有关；有大量供线程使用的寄存器的处理器与较简单的处理器相比，将需要更多刷新。实际上，这段代码在许多环境上执行了很长时间，而没有出现问题。只是在尝试每个线程有大量寄存器可用、基于 SPARC 的大型机时，才出现了问题。

这是不是就意味着，在较为小型的环境中，不太可能会遇到寄存器刷新相关的问题？或许是。但是正如多核 CPU 在简单笔记本上都已经司空见惯，配备了更多缓存和寄存器的更复杂的 CPU 也会越来越常见，像这样的隐藏问题将会暴露出来。

快速小结

1. 线程同步有两个性能方面的代价：限制了应用的可伸缩性，以及获取锁是有开销的。
2. 同步的内存语义、基于 CAS 的设施和 volatile 关键字对性能可能会有很大的影响，特别是在有很多寄存器的大型机上。

9.3.2 避免同步

如果同步可以完全避免，那加锁的损失就不会影响应用的性能。有两种一般性的方式可以应对。

其一是在每个线程中使用不同的对象，这样访问对象时就不存在竞争了。为保证线程安全，很多 Java 对象是同步的，但是它们未必需要共享。Random 类就是这样；第 12 章有一个 JDK 中的例子，即使用线程局部变量开发了一个新类，以避免 Random 类中的同步。

另一方面，很多 Java 对象创建的成本很高，或者是会占用大量内存。以 NumberFormat 类为例：这个类的实例不是线程安全的，因为需要构建一个 java.util.Locale 实例，以满足国际化需求，这使得构建新对象的成本非常高。一个程序用一个共享的全局 NumberFormat 实例也行得通，但是对这个共享对象的访问需要同步。

相反，更好的模式是使用 ThreadLocal 对象：

```
public class Thermometer {
    private static ThreadLocal<NumberFormat> nfLocal = new ThreadLocal<>() {
        public NumberFormat initialValue() {
            NumberFormat nf = NumberFormat.getInstance();
            nf.setMinumumIntegerDigits(2);
```

```
            return nf;
        }
    }
    public String toString() {
        NumberFormat nf = nfLocal.get();
        nf.format(...);
    }
}
```

通过使用一个线程局部变量，总的对象数得到了限制（使对 GC 的影响最小化），而且每个对象都不会受制于线程竞争。

其二是使用基于 CAS 的替代方案。在某种意义上，这不像是避免同步，更像是解决不同的问题。但是在这个背景之下，它也可以减少同步带来的性能损失，会得到同样的效果。

基于 CAS 的保护和传统的同步之间，其差别看上去正适合用微基准测试来测量：编写比较基于 CAS 的操作和传统同步方法的代码应该并不繁琐。比如，JDK 支持很方便地使用基于 CAS 保护的计数器：AtomicLong 及类似的类。微基准测试可以将使用了基于 CAS 包含的代码与传统的同步做对比。例如如下代码：

```
AtomicLong al = new AtomicLong(0);
public long doOperation() {
    return al.getAndIncrement();
}
```

对比以下代码：

```
    private volatile long al = 0;
    public synchronized doOperation() {
        return al++;
    }
```

结果表明，使用微基准测试是行不通的。如果只有一个线程（也就不存在竞争的可能性），使用上述代码所做的微基准测试可以合理地估算两种方案在无竞争环境下的代价（第 12 章也引用了这个测试结果）。但是对于存在竞争的环境，没法提供任何信息（而且如果代码不会存在竞争，那么一开始就不需要考虑线程安全了）。

仍然使用前面的两个代码片段，构造一个使用两个线程的微基准测试，会发现在共享资源上存在极大的竞争。这也并非现实中的情况：在实际的应用中，两个线程总是同步访问共享资源的情形不大可能出现。加入更多线程只是引入了更多并不现实的竞争情况。

竞争与 volatile 变量

开发者有时会考虑使用 volatile 变量来减少同步，进而减少其应用中的竞争。结果是，对 volatile 变量的同步写会非常缓慢。

本章前面使用 ForkJoinPool 的例子包含着一个专门设计的循环，即通过向一个 volatile 变量写无意义的值来消耗大量 CPU 周期：

```
for (int j = 0; j < d.length - i; j++) {
    for (int k = 0; k < 100; k++) {
```

```
            dummy = j * k + i; // dummy是一个volatile变量,因此发生了多次写操作
            d[i] = dummy;
        }
    }
```

dummy 被定义为这段代码所在类的一个实例变量，尽管例子中有 4 个线程同步执行，但是它们操作的是这个类的不同实例。因此，并不存在围绕 dummy 变量的竞争，例子中的测试在 16 秒内就完成了。

然而，如果将 dummy 修改为 static 变量，事情就会发生变化。现在有多个线程同时访问这个 volatile 变量，同样的测试需要 209 秒。

如第 2 章所讨论的，微基准测试容易大大高估所测问题中同步瓶颈的影响。希望这里的讨论也能说明这一点。如果将本节中的代码用在真实的应用中，可以更现实地对比两种方案的得失。

在通常情况下，在比较基于 CAS 的设施和传统的同步时，可以使用如下指导原则。

- 如果访问的是不存在竞争的资源，那么基于 CAS 的保护要稍快于传统的同步（虽然完全不使用保护会更快）。
- 如果访问的资源存在轻度或适度的竞争，那么基于 CAS 的保护要快于传统的同步（而且往往是快得多）。
- 随着所访问资源的竞争越来越剧烈，在某一时刻，传统的同步就会成为更高效的选择。在实践中，这只会出现在运行着大量线程的非常大型的机器上。
- 当被保护的值有多个读取，但不会被写入时，基于 CAS 的保护不会受竞争的影响。

Java 8 和存在竞争时的原子类

java.util.concurrent.atomic 包中的类使用了基于 CAS 的原语，而非传统的同步。因此，与编写一个同步方法来增加某个 long 变量相比，至少在对 CAS 原语的竞争非常剧烈之前，那些类（比如 AtomicLong 类）的性能往往要更好。

Java 8 引入了很多类来解决这个问题：原子的加法器（Adder）和累加器（Accumulator，比如 LongAdder 类）。与传统的原子类相比，这些类的可伸缩性更好。当多个线程更新某个 LongAdder 实例时，这个类可以独立保存每个线程所做的更新。这意味着，这些线程不需要等待其他线程完成其操作；相反，操作值本质上会被保存到一个数组中，每个线程都可以快速返回。之后，当某个线程尝试获取当前值时，操作值会被加起来或是累加起来。

在竞争很少或者没有竞争的情况下，值会随着程序的运行而累加，加法器的行为和传统的原子类一样。在竞争非常剧烈的情况下，更新会更快，不过实例会使用更多内存来保存值的数组。这种情况下，获取某个值也会稍微慢些，因为必须处理数组中所有挂起的值。不过，即使在竞争非常剧烈的情况下，与传统的原子类相比，这些新类也通常会表现得更好。

最后，还是要对代码所运行的真实生产环境做大量的测试，这是什么都代替不了的：只有这时，我们才能明确地说，到底某个特定方法的哪个实现会更好。不过即使是这样的真实情况，得出的判断也仅适用于当时那些条件。

快速小结

1. 避免对同步对象的竞争，是缓解同步对性能影响的有效方式之一。
2. 线程局部变量不会受竞争之苦；对于保存实际不需要在多个线程间共享的同步对象，它们非常理想。
3. 对于确实需要共享的对象，基于 CAS 的工具也是避免传统的同步的方式之一。

9.3.3 伪共享

在同步可能的影响方面，有一点很少被讨论到，就是伪共享（false sharing）。在多线程程序中，这个问题过去相当隐蔽，但是随着多核机器成为标配，很多同步性能问题更明显地浮出水面了。伪共享就是一个越来越重要的问题。

伪共享之所以会出现，跟 CPU 处理其高速缓存的方式有关。考虑一个简单类中的数据：

```
public class DataHolder {
    public volatile long l1;
    public volatile long l2;
    public volatile long l3;
    public volatile long l4;
}
```

这里的每个 `long` 值都保存在毗邻的内存位置中；比如，`l1` 可能保存在 0xF20 这个内存位置。那么 `l2` 会保存在 0xF28，`l3` 在 0xF2C，以此类推。当程序要操作 `l2` 时，会有一大块内存被加载到当前所用的某个 CPU 核上，比如说，从 0xF00 到 0xF80 的 128 字节。如果有第 2 个线程要操作 `l3`，则会加载同样一段内存到另一个核的缓存行（cache line）中。

大多数情况下，像这样加载邻接的值是有意义的：如果程序访问了对象中的某个特定实例变量，则很有可能会访问邻接的实例变量。如果这些实例变量被加载到当前核的高速缓存中，内存访问就非常快，这是很大的性能优势。

这种模式的缺点是，当程序更新本地缓存中的某个值时，当前的核必须通知其他所有核：这个内存被修改了。其他核必须作废其缓存行，并重新从内存中加载。

我们来看看，如果有多个线程大量使用 `DataHolder` 类会发生什么：

```
public class ContendedTest extends Thread {
    private static class DataHolder {
        private volatile long l1 = 0;
        private volatile long l2 = 0;
        private volatile long l3 = 0;
        private volatile long l4 = 0;
    }
    private static DataHolder dh = new DataHolder();
    private static long nLoops;
```

```
        public ContendedTest(Runnable r) {
            super(r);
        }

        public static void main(String[] args) throws Exception {
            nLoops = Long.parseLong(args[0]);
            ContendedTest[] tests = new ContendedTest[4];
            tests[0] = new ContendedTest(() -> {
                    for (long i = 0; i < nLoops; i++) {
                            dh.l1 += i;
                    }
            });
            tests[1] = new ContendedTest(() -> {
                    for (long i = 0; i < nLoops; i++) {
                            dh.l2 += i;
                    }
            });
            //……tests[2]和tests[3]类似……
            long then = System.currentTimeMillis();
            for (ContendedTest ct : tests) {
                ct.start();
            }
            for (ContendedTest ct : tests) {
                ct.join();
            }
            long now = System.currentTimeMillis();
            System.out.println("Duration: " + (now - then) + " ms");
        }
    }
```

结果并非如此：当一个特定的线程在其循环中写 volatile 值时，其他每个线程的缓存行都会被作废，内存值必须重新加载。结果如表 9-8 所示，性能随着线程数增多而变差了。

表9-8：存在伪共享时，对1 000 000个值求和的时间

线程数	消耗时间（秒）
1	7.1
2	52.1
3	91.0
4	128.3

严格来讲，伪共享未必会涉及同步（或 volatile）变量：不论何时，CPU 缓存中有任何数据被写入了，其他保存了同样范围数据的缓存都必须作废。然而，切记 Java 内存模型要求数据只是在同步原语（包括 CAS 和 volatile 构造）结束时必须写入主内存。所以这种情况是最常见的。在这个例子中，如果 long 变量不是 volatile 的，那么编译器会将这些值放到寄存器中，不管有多少个线程，测试将在大约 7.1 秒内执行完毕。

很明显，这是个极端的例子，但是它提出了一个问题：如何检测并纠正伪共享？遗憾的是，并没有清晰、完整的答案。在第 3 章所讨论的标准工具集中，没有哪个能解决伪共享，因为这需要与处理器架构相关的非常专门的知识。

如果幸运的话，目标处理器的厂商会提供用于诊断伪共享的工具。比如，Intel 就有一个叫作 VTune 的程序，可以通过检查缓存未命中事件来检测伪共享。特定的原生分析器（profiler）可能会提供给定代码行的每指令周期数（Cycles Per Instruction，CPI）的相关信息；如果某个循环内的一条简单指令的 CPI 非常高，可能预示着代码正在等待将目标内存的信息重新加载到 CPU 缓存中。

另外，检测伪共享还需要一些直觉和实验。如果正常的分析表明，某个特定循环耗时惊人，则需要检查这个循环，看看是否有可能循环内有多个线程正在访问非共享变量。（即便很多人认为性能调优更像是艺术而非科学，Intel VTune 手册也写道："避免伪共享的主要手段就是代码检查"。）

要阻止伪共享，需要对代码做些修改。理想的情况是所涉及的变量不会频繁写入。在前面的例子中，计算可以使用局部变量进行，只有最终结果才写回到 `DataHolder` 变量。如果随后的写入次数比较少，就不太可能出现对缓存行的竞争，即使所有 4 个线程在循环结束时同时更新其结果，也不会影响性能。

第 2 个可能的方案是填充（padding）相关变量，以免其被加载到相同的缓存行中。如果目标 CPU 有个 128 字节的缓存，那么像下面这样填充可能会有效果（也可能没有）：

```
public class DataHolder {
    public volatile long l1;
    pubilc long[] dummy1 = new long[128 / 8];
    public volatile long l2;
    pubilc long[] dummy2 = new long[128 / 8];
    public volatile long l3;
    pubilc long[] dummy3 = new long[128 / 8];
    public volatile long l4;
}
```

像这样使用数组或许行不通，因为 JVM 可能会重新安排那些实例变量的布局，以便所有的数组紧挨在一起，于是所有的 `long` 变量就仍然会紧挨着了。使用基本类型的值来填充该结构，行之有效的可能性更大，但是考虑到所需的变量数目，并不现实。

使用填充来防止伪共享还有其他问题。填充的大小很难预测，因为不同 CPU 的缓存大小也不同。而且填充很明显会增大问题中的实例，这对垃圾收集器影响很大（当然也取决于所需的实例数）。不过，如果没有算法上的改进方案，填充数据有时会具有明显的优势。

@Contended 注解

Java 8 有个新特性，即能够减少指定字段（JEP 142）上的竞争。其实现方式是使用一个新的注解（`@sun.misc.Contended`）来标记应该由 JVM 自动填充的变量。

这个注解所属的包非常重要：尽管这是一个 JDK 增强提案（JDK Enhancement Proposal，JEP）特性，但它主要是供 JVM 自身使用。这个注解是否会进入未来的版本（即使会，其行为是否会保持不变），并没有保证。

如果不能通过其他任何手段解决伪共享，可以考虑使用该注解。一个好处是，因为 JVM 了解其运行所在 CPU 的架构，所以可以自动算出需要填充的大小；然而当 AMD

（或是无论哪家厂商）推出配备了新的缓存行大小的新处理器时，较老的 JVM 只能猜测需要填充的大小是多少。和所有的填充解决方案一样，使用这个注解会大大增加目标实例的大小，所以要谨慎使用。

默认情况下，除了 JDK 内部的类，JVM 会忽略该注解。要支持应用代码使用该注解，应该使用 `-XX:-RestrictContended` 标志，它默认为 `true`（意味着该注解仅限于 JDK 类使用）。另一方面，要关掉 JDK 中的自动填充，应该设置 `-XX:-EnableContended` 标志，它也默认为 `true`。这将减小 `Thread` 和 `ConcurrentHashMap` 类的大小。

快速小结

1. 对于会频繁地修改 `volatile` 变量或退出同步块的代码，伪共享对性能影响很大。
2. 伪共享很难检测。如果某个循环看上去非常耗时，可以检查该代码，看看是否与伪共享出现时的模式相匹配。
3. 最好通过将数据移到局部变量中、稍后再保存来避免伪共享。作为一种替代方案，有时可以使用填充将冲突的变量移到不同的缓存行中。

9.4　JVM线程调优

JVM 的某些调优策略可以影响线程和同步的性能。

9.4.1　调节线程栈大小

当空间非常珍贵时，可以调节线程所用的内存。每个线程都有一个原生栈，操作系统用它来保存该线程的调用栈信息（比如，`main()` 方法调用了 `calculate()` 方法，而 `calculate()` 方法又调用了 `add()` 方法，栈会把这些信息记录下来）。

不同的 JVM 版本，其线程栈的默认大小也有所差别，具体如表 9-9 所示。一般而言，如果在 32 位 JVM 上有 128 KB 的栈，在 64 位 JVM 上有 256 KB 的栈，很多应用实际就可以运行了。如果这个值设置得太小，潜在的缺点是，当某个线程的调用栈非常大时，会抛出 `StackOverflowError`。

表9-9：种JVM的默认栈大小

操作系统	32位	64位
Linux	320 KB	1 MB
Mac OS	N/A	1 MB
Solaris Sparc	512 KB	1 MB
Solaris X86	320 KB	1 MB
Windows	320 KB	1 MB

在 64 位的 JVM 中，除非物理内存非常有限，并且较小的栈可以防止耗尽原生内存，否则没有理由设置这个值。另一方面，在 32 位的 JVM 上，使用较小的栈（比如 128 KB）往往

是个不错的选择，因为这样可以在进程空间中释放部分内存，使得 JVM 的堆可以大一些。

耗尽原生内存

没有足够的原生内存来创建线程，也可能会抛出 `OutOfMemoryError`。这意味着可能出现了以下 3 种情况之一。

1. 在 32 位的 JVM 上，进程所占空间达到了 4 GB 的最大值（或者小于 4 GB，取决于操作系统）。
2. 系统实际已经耗尽了虚拟内存。
3. 在 Unix 风格的系统上，用户创建的进程数已经达到配额限制。这方面单独的线程会被看作一个进程。

减少栈的大小可以克服前两个问题，但是对第三个问题没什么效果。遗憾的是，我们无法从 JVM 报错看出到底是哪种情况，只能在遇到错误时依次排查。

要改变线程的栈大小，可以使用 `-Xss=N` 标志（例如 `-Xss=256k`）。

快速小结

1. 在内存比较稀缺的机器上，可以减少线程栈大小。
2. 在 32 位的 JVM 上，可以减少线程栈大小，以便在 4 GB 进程空间限制的条件下，稍稍增加堆可以使用的内存。

9.4.2 偏向锁

当锁被争用时，JVM（和操作系统）可以选择如何分配锁。锁可以被公平地授予，每个线程以轮转调度方式（round-robin）获得锁。还有一种方案，即锁可以偏向于对它访问最为频繁的线程。

偏向锁背后的理论依据是，如果一个线程最近用到了某个锁，那么线程下一次执行由同一把锁保护的代码所需的数据可能仍然保存在处理器的缓存中。如果给这个线程优先获得这把锁的权利，缓存命中率可能就会增加。如果实现了这点，性能会有所改进。但是因为偏向锁也需要一些簿记信息，故有时性能可能会更糟。

特别是，使用了某个线程池的应用（包括大部分应用服务器），在偏向锁生效的情况下，性能会更糟糕。在那种编程模型下，不同的线程有同等机会访问争用的锁。对于这些类应用，使用 `-XX:-UseBiasedLocking` 选项禁用偏向锁，会稍稍改进性能。偏向锁默认是开启的。

9.4.3 自旋锁

在处理同步锁的竞争问题时，JVM 有两种选择。对于想要获得锁而陷入阻塞的线程，可以让它进入忙循环，执行一些指令，然后再次检查这个锁。也可以把这个线程放入一个队列，在锁可用时通知它（使得 CPU 可供其他线程使用）。

如果多个线程竞争的锁的被持有时间较短，那忙循环（所谓的线程自旋）就比另一个方案快得多。如果被持有时间较长，则让第二个线程等待通知会更好，而且这样第三个线程也

有机会使用 CPU。

JVM 会在这两种情况间寻求合理的平衡，自动调整将线程移交到待通知队列中之前的自旋时间。有些参数可以调整自旋时间，但大部分是实验性的，都有可能会发生变化，即使是极小的版本更新。

如果想影响 JVM 处理自旋锁的方式，唯一合理的方式就是让同步块尽可能短；当然不管什么情况，都是应该这么做的。这样可以限制与程序功能没有直接关系的自旋的量，也降低了线程进入通知队列的机会。

> **UseSpinning 标志**
>
> 之前的 Java 版本支持一个 -XX:+UseSpinning 标志，该标志可以开启或关闭自旋锁。在 Java 7 及更高版本中，这个标志已经没用了：自旋锁无法禁用。不过考虑到向后兼容，Java 7 到 7u40 这些版本的命令行参数仍然接受该标志，但是不执行任何操作。有点奇怪的是，这个标志的默认值会报告为 false，即使自旋锁一直在发挥作用。
>
> 从 Java 7u40（以及 Java 8 中）开始，Java 不再支持该标志，使用这个标志会报错。

9.4.4 线程优先级

每个 Java 线程都有一个开发者定义的优先级，这是应用提供给操作系统的一个线索，用以说明特定线程在其眼中的重要程度。如果有不同线程处理不同任务，你可能会认为，可以以让其他任务在优先级较低的线程上运行为代价，使用线程优先级来改进特定任务的性能。遗憾的是，实际不会这么有用。

操作系统会为机器上运行的每个线程计算一个“当前”（current）优先级。当前优先级会考虑 Java 指派的优先级，但是还会考虑很多其他的因素，其中最重要的一个是：自线程上次运行到现在所持续的时间。这可以确保所有的线程都有机会在某个时间点运行。不管优先级高低，没有线程会一直处于“饥饿”状态，等待访问 CPU。

这两个因素之间的平衡会随操作系统的不同而有所差异。在基于 Unix 的系统上，整体优先级的计算主要取决于线程上次运行到现在所持续的时间，Java 层指定的优先级影响微乎其微。在 Windows 系统上，在 Java 层指定的优先级较高的线程，往往会比优先级较低的线程运行更久；但即便优先级较低，那些线程也会得到相对公平的执行时间。

不过，不管是哪种情况，我们都不能依赖线程的优先级来影响其性能。如果某些任务比其他任务更重要，就必须使用应用层逻辑来划分优先级。

在某种程度上，可以通过将任务指派给不同的线程池并修改那些池的大小来解决。第 10 章有一个这样的例子。

9.5 监控线程与锁

在对应用中的线程和同步的效率作性能分析时，有两点需要注意：总的线程数（既不能太大，也不能太小）和线程花在等待锁或其他资源上的时间。

9.5.1 查看线程

几乎所有的 JVM 监控工具都提供了线程数（以及这些线程在干什么）相关的信息。像 `jconsole` 这样的交互式工具还能显示 JVM 内线程的状态。在 `jconsole` 的 Threads 面板上，可以实时观察程序执行期间线程数的增减。图 9-2 是一个例子。

在某个时间点，应用（NetBeans）最多使用了 45 个线程。图中刚开始有一个爆发点，最多会使用 38 个线程，后来线程数稳定在 30 到 31 之间。`jconsole` 可以打印每个单独线程的栈信息；如图所示，Java2D Disposer 线程正在某个引用队列的锁上等待。

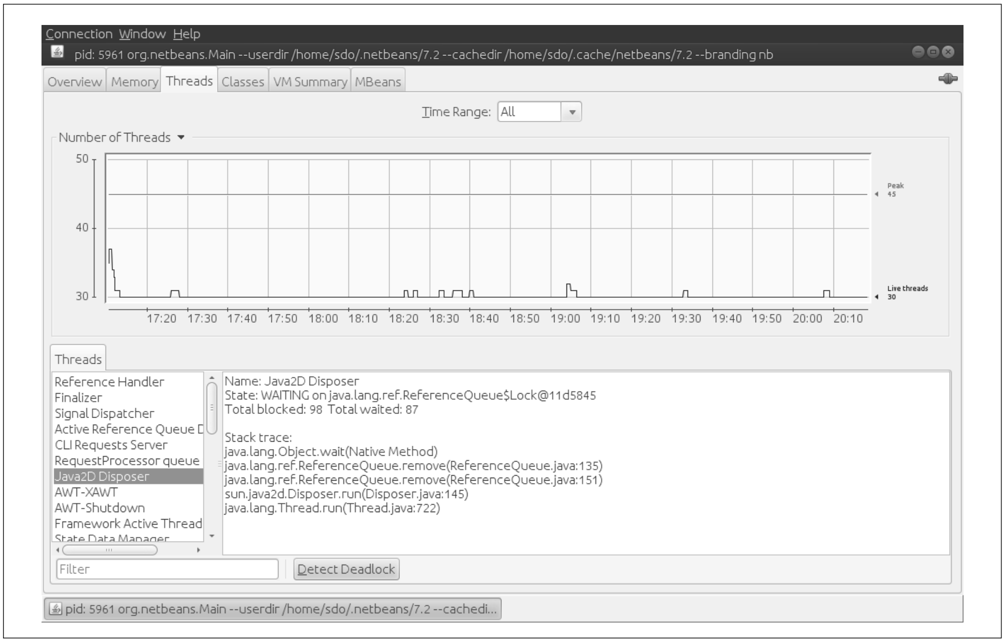


图 9-2：JConsole 中的活跃线程视图

9.5.2 查看阻塞线程

如果想了解应用中有什么线程在运行这类高层视图，实时线程监控会很有用，但至于那些线程在做什么，实际上没有提供任何数据。要确定线程的 CPU 周期都耗在哪儿了，则需要使用分析器（profiler），第 3 章曾讨论过。利用分析器可以很好地观察哪些线程在执行。而且分析器一般非常成熟，可以指出那些能够通过更好的算法、更好的代码选择来加速整体执行效果的代码区域。

诊断阻塞的线程更为困难，尽管这类信息对应用的整体执行而言往往更为重要，特别是当代码运行在多 CPU 系统上，但没有利用起所有可用的 CPU 时。一般有三种执行此类诊断的方法。方法之一还是使用分析器，因为大部分分析工具都会提供线程执行的时间线信息，这就可以看到线程被阻塞的时间点。第 3 章也给出了一个例子。

1. 被阻塞线程与JFR

要了解线程是何时被阻塞的，迄今为止最好的方式是使用可以窥探 JVM 内部、并且可以在较低的层次确定线程被阻塞时间的工具。Java 飞行记录器（Java Flight Recorder，JFR）就是一款这样的工具，第 3 章已经介绍过。我们可以深入到 JFR 捕获的事件中，并寻找那些引发线程阻塞的事件（比如等待获取某个 Monitor，或是等待读写 Socket，不过写的情况较为少见）。

借助 JMC 的直方图面板可以很方便地查看这些事件，如图 9-3 所示。

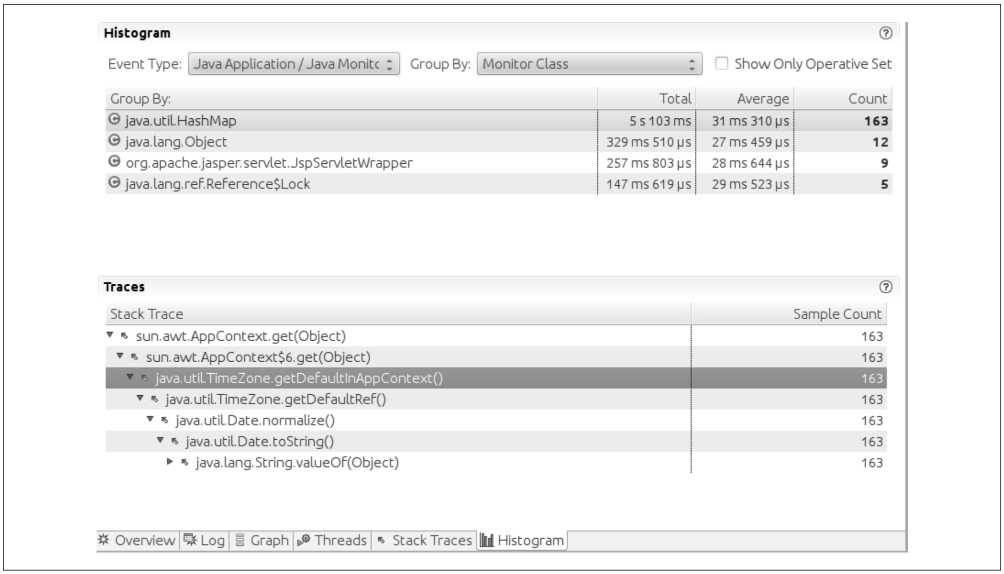


图 9-3：JFR 中被某个 Monitor 阻塞的线程

在这个示例中，与 `sun.awt.AppContext.get()` 方法中的 `HashMap` 关联的锁被竞争了 163 次（超过 66 秒），使得所测量的请求响应时间平均增加了 31 毫秒。栈轨迹表明竞争源于 JSP 写 `java.util.Date` 对象的方式。要改进这段代码的可伸缩性，可以使用线程局部的日期格式化对象，而不是简单地调用日期对象的 `toString()` 方法。

从直方图中选择阻塞事件，然后检查调用代码，这个流程适合任何阻塞事件；这款与 JVM 紧密集成的工具使这一流程就成为可能。

2. 被阻塞线程与JStack

如果没有商用的 JVM 可用，替代方案之一是从程序中拿到大量的线程栈并加以检查。`jstack`、`jcmd` 和其他工具可以提供虚拟机中每个线程状态相关的信息，包括线程是在运行、等待锁还是等待 I/O 等。对于确定应用中正在进行的是什么，这可能非常有用，不过输出中也有很多我们不需要的。

在查看线程栈时，有两点需要注意。第一，JVM 只能在特定的位置（safepoint，安全点）转储出一个线程的栈。第二，每次只能针对一个线程转储出栈信息，所以可能会看到彼

此冲突的信息：比如两个线程持有同一个锁，或者一个线程正在等待的锁并未被其他线程持有。

> ### JStack 分析器
> 人们很容易认为，连续快速地抓取多个栈转储信息，就能将其用作一个简单快速的分析器。毕竟，采样分析器本质上就是这么工作的：周期性地探测线程的执行栈，基于这些信息推断在方法上花了多少时间。但是在安全点和不一致的快照之间，这么做不是很有效；通过查看这些线程栈，有时可以从较高的层次上大概获知执行成本较高的方法，但是一款真正的分析器提供的信息要精确得多。

从线程栈可以看出线程阻塞的严重程度（因为阻塞的线程已经在某个安全点上）。如果有连续的线程转储信息表明大量的线程阻塞在某个锁上，那么就可以断定这个锁上有严重的竞争。如果有连续的线程转储信息表明大量的线程在阻塞等待 I/O，则可以断定需要优化正在进行的 I/O 读操作（比如，如果是数据库调用，应该优化 SQL 执行，或者是优化数据库本身）。

在本书的在线示例中，有一个比较基本的 `jstack` 输出解析器，可以从一个或多个线程转储中总结出所有线程的状态。`jstack` 的输出有个问题，即不同版本之间可能会有变化，所以开发一个健壮的解析器比较困难。不能保证这个解析器可以不加修改地应用于你所使用的特定的 JVM。

`jstack` 解析器的基本输出像下面这样：

```
% jstack pid > jstack.out
% java ParseJStack jstack.out
[Partial output...]
Threads in start Running
    8 threads in java.lang.Throwable.getStackTraceElement(Native
Total Running Threads: 8

Threads in state Blocked by Locks
    41 threads running in
        com.sun.enterprise.loader.EJBClassLoader.getResourceAsStream
        (EJBClassLoader.java:801)
Total Blocked by Locks Threads: 41

Threads in state Waiting for notify
    39 threads running in
        com.sun.enterprise.web.connector.grizzly.LinkedListPipeline.getTask
        (LinkedListPipeline.java:294)
    18 threads running in System Thread
Total Waiting for notify Threads: 74

Threads in state Waiting for I/O read
    14 threads running in com.acme.MyServlet.doGet(MyServlet.java:603)
Total Waiting for I/O read Threads: 14
```

解析器聚合了所有的线程，可以显示处于各种状态的线程分别有多少。8 个线程正在运行（它们碰巧正在获取栈轨迹信息，这个操作成本非常高，最好避免）。

41 个线程被某个锁阻塞了。所报告的方法是栈轨迹中第一个非 JDK 方法，在这个例子中是 GlassFish 的 `EJBClassLoader.getResourceAsStream()`。下一步就是考虑栈轨迹信息，搜索这个方法，看看线程是阻塞到什么资源上了。

在这个例子中，所有线程都被阻塞了，在等待读取同一个 JAR 文件；这些线程的栈轨迹表明，所有调用都来自实例化新 SAX 实例的操作。如第 10 章所讨论的，SAX 解析器可以通过列出应用 JAR 文件中 manifest 文件内的资源来动态定义，这意味着 JDK 必须搜索整个类路径来寻找那些条目，直到找到应用想使用的一个（或者是找不到，回到系统解析器）。因为读取这个 JAR 文件需要一个同步锁，所以所有尝试创建一个解析器的线程最终都会竞争同一个锁，这会极大影响应用的吞吐量。（第 10 章建议设置 `-Djavax.xml.parsers.SAXParserFactory` 属性来避免这些查找，原因就在于此。）

更重要的一点是，大量被阻塞的线程会成为影响性能的问题。不管阻塞的根源是什么，都要对配置或应用加以修改，以避免之。

等待通知的线程又是什么样的情况呢？那些线程在等待其他事件发生。它们往往是在某个池中，等待任务就绪（比如，上面输出中的 `getTask()` 方法在等待请求）这类通知。系统线程会在处理像 RMI 分布式 GC 或 JMX 监控这样的事情，它们以栈中只有 JDK 类这类线程的形式出现在 `jstack` 的输出中。这些条件不一定表明有性能问题；对这些线程而言，等待通知是正常现象。

如果线程正在进行的是阻塞式 I/O 读取（通常是 `socketRead0()` 方法），也会导致问题。这也会影响吞吐量：线程正在等待某个后端资源回复其请求。这时候应该检查数据库或其他后端资源的性能。

快速小结

1. 利用系统提供的线程基本信息，可以对正在运行的线程的数目有个大致了解。
2. 就性能分析而言，当线程阻塞在某个资源或 I/O 上时，能够看到线程的相关细节就显得比较重要。
3. JFR 使得我们可以很方便地检查引发线程阻塞的事件。
4. 利用 `jstack`，一定程度上可以检查线程是阻塞在什么资源上。

9.6 小结

理解线程如何运作，可以获得很大的性能优势。不过就线程的性能而言，其实没有太多可以调优的：可以修改的 JVM 标志相当少，而且那些标志的效果也很有限。

相反，较好的线程性能是这么来的：遵循管理线程数、限制同步带来的影响的一系列最佳实践原则。借助适当的剖析工具和锁分析工具，可以检查并修改应用，以避免线程和锁的问题给性能带来负面影响。

第10章

Java EE性能调优

本章关注的是 Java EE（特别是 Java EE 6 和 7），涵盖了 JSP、servlet 和 EJB 3.0 会话 Bean——因为 EJB 3.0 实体 Bean（Java 持久化 API 实体，即 JPA）并不限定于 Java EE 技术（第 11 章将深入讨论），所以本章没有涵盖。

10.1 Web容器的基本性能

Java EE 应用服务器性能的关键是 Web 容器，它通过基本的 servlet 和 JSP 页面处理 HTTP 请求。

有些基本的途径可以改善 Web 容器的性能，改进的具体方法因 Java EE 实现的不同而有所不同，但一些概念可以适用于所有服务器。

减少输出

减少服务器产生的结果输出可以加快 Web 页面返回到浏览器的速度。

减少空格

在 servlet 代码中调用 `PrintWriter` 时不要写入多余的空格，因为空格在网络上传输时同样需要时间（而且，相对于代码的处理，网络传输时间更为重要）。你应该用 `print()` 而不是 `println()`，主要是为了避免在返回结果的 HTML 中写入制表符或空格。虽然这确实会使有些人查看 Web 页面源代码时看不清结构，但如果他们真对源代码感兴趣，总会使用 XML 或 HTML 编辑器。也可以让内部 QA 或者性能优化小组来处理空格。毫无疑问，结构化的页面源代码可以简化调试，但为了改善应用的响应时间，我最后还得把它载入格式编辑器以去除多余的空格。绝大多数应用服务器都可以自动去除 JSP 页面中的空格。比如 Tomcat（以及基于 Tomcat 的开源 Java EE 服务器）中的 `trimSpaces` 指令，可以将 JSP 页面每行的前后空格都去掉。所以开发和维护 JSP 页面时可以有适当的

（至少对人类来说是如此）缩进，而不用担心会在网络上传输不必要的空格。

合并 CSS 和 JavaScript 资源

对于开发者来说，把 CSS 保存在独立的文件中是有意义的，也更容易维护。对于 JavaScript 来说也是如此。但使用这些资源时，传输一个大文件的效率比传输几个小文件要高。Java EE 没有这方面的标准，而且绝大多数应用服务器也无法自动处理，不过有些开发工具可以帮助你合并这些资源。

压缩输出

从用户角度来看，执行 Web 请求的最长时间通常是服务器将 HTML 发回浏览器所需的时间。但由于客户端（模拟浏览器）到服务器的性能测试通常在快速局域网中进行，所以这个时间通常并不是最长的。虽然真实用户可能在“快速”广域网中，但仍然要比你实验室里的机器之间的 LAN 慢一个数量级。大多数应用服务器在将数据发回浏览器时都有压缩机制：HTML 数据压缩发送给浏览器，内容类型（content type）为 `zip` 或 `gzip`。这只有在初始请求指明浏览器支持压缩时才行得通。所有的现代浏览器都支持该特性。开启压缩要求服务器有更多的 CPU 周期，但通常数据量越小，网络传送的时间也越少，从而整体性能就会越高。然而与本节讨论的其他优化不同，它并不总能提高性能。本节后面的例子表明，在 LAN 开启压缩时，性能可能会下降。应用发送很小的页面时也会如此（尽管大多数应用服务器允许只有输出大于某个特定尺寸时才压缩）。

不要使用 JSP 动态编译

默认情况下，大多数 Java EE 应用服务器允许 JSP 页面动态更改：JSP 文件可以随时编辑（无论部署在哪里），而这些变化将在下次访问页面时起作用。这在开发新 JSP 时非常有用，但因为每次访问 JSP 时，服务器都要通过检查文件的最后修改日期来判断是否需要重新加载，所以在生产环境中就会拖慢服务器。这通常被称为开发模式，应该在生产和性能测试时关闭。

字符串是否应该预编码？

应用服务器在字符转换上要花费大量时间：从 Java 的 `String` 对象（以 UTF-16 格式保存）转换成客户端所需要的字节数组。许多这样的字符串总是相同的。Web 页面的 HTML 字符串并不会总随着数据发生变动（如果发生了，它们也仍然是从字符串常量集合中获取的）。

这些字符串是否应该预先编码成字节数组以便可以重用？答案取决于应用服务器和应用本身。

JSP 页面的 HTML 字符串由应用服务器所写。字符串是否预先编码取决于服务器：有些服务器会对此提供一个选项，有一些则是自动执行的。

在 servlet 中，这些字符串可以预编码，然后用 `ServletOutputStream` 的 `write()` 通过网络发送，不要用 `PrintWriter` 的 `print()`。不过动态数据仍然要用 `print()` 才能正确编码。

（你可以从 header 中找到目标编码，然后对字符串编码，但这种方法相对容易出错。）

应用服务器实现这些输出接口以及在其内部缓存这些数据的方式有很大差别。对一些服务器来说，混用 servlet 的输出流（output stream）和它的小伙伴 print writer 会导致频繁刷新网络缓存。从性能优化角度看，频繁刷新缓存是非常昂贵的操作——比重新编码这些数据更昂贵。与此类似，对一大块数据进行编码的代价通常不会比一小块数据高很多：最主要的代价是建立到编码器的调用。因此，对小段动态数据来说，频繁地编码及发送编码后的字节数组会拖慢应用：多次调用编码器所花费的时间，比一次调用编码所有的东西（包括静态数据）要长。

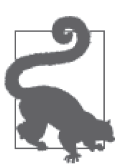
代码的预编码在某些情况下有一定作用，但要视情况而定。

与测试相比，这些优化措施实际运行中的性能会有很大差别。表 10-1 显示了可能会出现的结果。测试中所用股票历史 servlet 产生的输出比较长，获取的数据范围有 10 年。所产生的结果是未经压缩和未去除空格的 HTML 页面，大约为 100 KB。为了将带宽的影响降至最低，测试只运行单个用户，思考时间为 100 毫秒，然后测量请求的平均响应时间。使用局域网时，测试通过 100 MB 的交换机在本地网络上运行；使用宽带时，测试在家里的电缆上运行（平均每秒 30 Mb 的下载速度）。使用本地咖啡店中的公共 WiFi 连接的广域网时——网速是相当不可靠的（表格中展示了历时 4 个小时的平均样本）。

表10-1：几种Web响应输出尺寸方面的优化在不同网络条件下的效果

所使用的优化	应用响应时间（局域网，毫秒）	应用响应时间（宽带，毫秒）	应用响应时间（公共WiFi，毫秒）
无	20	26	1003
去除空格	20	10	43
压缩输出结果	30	5	17

这张表强调了在应用的实际部署环境中进行测试的重要性。如果只在实验室环境中进行测试调优，那得到的一大半性能都是不太靠谱的。虽然这个例子中的测试实际运行在远程应用服务器上（使用公有云服务），但硬件模拟器可以模拟出实验室环境，控制所有相关的机器。（云服务机器也比局域网机器快；它们之间的机器数量无法直接进行比较。）



快速小结

1. 在 Java EE 应用所实际运行的网络基础设施上对它们进行测试。
2. 外部网络相对内部网络来说仍然是慢的。限制应用所写的数据量会取得很好的性能。

HTTP会话状态

关于 HTTP 会话状态有两个重要的性能提示。

1. HTTP会话状态的内存占用

请注意应用管理 HTTP 会话状态的方式。HTTP 会话数据通常存活时间很长，所以很容易塞满堆内存，也常常容易导致 GC 运行太频繁的问题。（此外，回想第 7 章中的内容，堆中

的存活数据越多，单次 GC 所用的时间也会越长。）

这个问题最好在应用层面解决：决定在 HTTP 会话中存储数据前需三思而后行。如果数据可以很容易地重建，最好就不要保存在会话状态中。此外，还需要留意会话数据保留的时长。应用会话数据保存的时长在 web.xml 文件中，默认值为 30 分钟：

```
<session-timeout>30</session-timeout>
```

会话数据保留的时间太长了——真的有用户在离开 29 分钟后再返回么？调低这个值可以显著缓解太多会话数据对堆内存造成的压力。

这部分是 Java EE 应用服务器的具体实现可以提供的帮助。虽然会话数据必须保留 30 分钟（或其他值），但数据没有必要保存在 Java 堆中。应用服务器可以（通过序列化）将会话数据移到磁盘或者远程缓存中——比如说，在空闲了 10 分钟之后进行。这可以释放应用服务器的堆内存空间，同时依然遵循保留应用状态 30 分钟（或其他值）的约定。如果用户 29 分钟之后回来了，那他的首次请求耗时会长一些，因为需要从磁盘读取状态，但在此期间的整体应用服务器性能会更好。

这也是测试时需要牢记的一个重要原则：面对应用的用户，什么样的会话管理是切实可预期的？他们是早上登录后一整天都使用该会话，还是来去很频繁，在服务器上留下大量的废弃会话，还是介于两者之间？无论答案是什么，都应该确保测试反映所期望的会话场景。否则生产服务器就会被错误调优，因为此时堆的使用完全不同于性能测试时的状况。

负载生成器有不同的会话管理方式，但一般来说，可以选择在测试的某个时间点开启一个新会话（可通过以下方式实现这一点，即关闭连接服务器的 socket 并且丢弃之前所有的 cookie）。本书所有的测试都使用 `fhb`，每个客户端线程的每轮测试维护一个线程。（不过，实际上 `fhb` 并没有创建新会话的选项，尽管通过 `faban` 中定制的驱动可以做到这点。）

2. HTTP会话状态的高可用（Highly available HTTP session state）

如果应用服务器在高可用（HA）配置下测试，那么必须留意服务器如何复制会话状态数据。应用服务器可以选择在每个请求中复制完整的会话状态，或者只在数据发生更改时复制。毫无疑问，第二种方法性能更高。同样，这是大多数应用服务器都支持的特性，不过不同供应商的设置不同。如何依据配置属性进行复制，请参考应用服务器文档。

不过，要使这个法子管用，开发人员必须遵循一定的规则来处理会话状态。特别是，应用服务器无法追踪已经存储在会话中的对象的变化。如果从会话中获取一个对象，然后改变它，必须调用 `setAttribute()` 方法让应用服务器知道那个对象的值发生了变化：

```
HttpSession session = request.getSession(true);
ArrayList<StockPriceHistory> al =
    (ArrayList<StockPriceHistory>) session.getAttribute("saveHistory");
al.add(……一些数据……);
session.setAttribute("saveHistory", al);
```

在单个（非复制）服务器上，末尾的那句 `setAttribute()` 并不是必需的：因为 `al` 已经在会话状态里了。如果省略了该调用，将来该会话中的所有请求都会发到服务器，一切都会正常工作了。

对于复制服务器来说，如果省略了该调用，会话会被复制到备份服务器上，请求会被备份服务器处理，应用可能会发现 `al` 数据没有发生变化。这是因为应用服务器“优化”了会话状态的处理，即只复制变化的数据到备份服务器上。没调用 `setAttribute()` 的话，应用服务器就不知道 `al` 发生了变化，所以执行完上述代码后不会复制它。

某种程度上来说，这是 Java EE 规范中的灰色区域。规范没有强制在这种情况下必须调用 `setAttribute()`，但每种 Java EE 应用服务器实际上都遵循这种惯例。对某些应用服务器来说，这是会话复制机制正常工作的唯一方式。而其他还有些应用服务器则允许配置数据复制的方式——包括每次调用时都复制所有的会话状态数据，所以即便应用不调用 `setAttribute()`，也能正常工作。虽然这种做法功能上没问题，但性能要比只在属性更改时复制要差很多。

这个事实的真正含义在于：一旦你更改了会话状态中的对象值，都应该调用 `setAttribute()`，并确保你的应用服务器配置成只复制更改的数据。

快速小结

1. 会话状态会对应用服务器的性能造成重大影响。
2. 尽可能少地在会话状态中保留数据，尽可能缩短会话的有效期，以减少会话状态对垃圾收集的影响。
3. 仔细查看应用服务器的调优规范，将非活跃的会话数据移出堆。
4. 开启会话高可用时，需要确保将应用服务器配置成只在状态属性发生变化时进行会话复制。

10.2 线程池

第 9 章深入介绍了线程池。Java EE 服务器则扩展了线程池的使用，同时第 9 章所介绍的关于如何正确调整线程池大小的所有内容也都适用于应用服务器。

应用服务器通常不只有一个线程池。一个线程池通常用来处理 servlet 的请求，另一个则用来处理远程 EJB 的请求，第三个则可以处理 Java Message Service（JMS）请求。也有些应用服务器允许每类请求可以有多个线程池：比如，同样是 servlet 请求，但 URL 不同，可以由单独的线程池处理，或同样是远程 EJB 请求，但调用的 EJB 不同，也可以由单独的线程池处理。

应用服务器中的线程池可以依据不同的请求量分成若干优先级。以运行在四 CPU 机器上的应用服务器为例，假设它的 HTTP 线程池有 12 个线程，EJB 线程池有 4 个。所有线程都会争抢 CPU，但当所有线程都跑满的时候，servlet 请求使用 CPU 的机会比 EJB 请求多两倍。实际上 servlet 线程池的优先级为 3x。

不过这里也有些限制。这些线程池没有办法单独设置，所以只有在没有待处理的 servlet 请求时，才会处理 EJB 请求。只要 EJB 线程池中还有可用线程，这些线程就会共同争用 CPU，无论 servlet 线程池有多忙。

类似地，当服务器因为其他原因空闲时，也请注意，不要将线程池的大小设置成低于预期

的工作负荷。如果四 CPU 机器的 JMS 池只配置 3 个线程，那只处理 JMS 请求的话就无法充分利用 CPU。为了弥补这样的浪费，所有线程池的大小都能相应增大，不过这可能会使你的机器运行了太多的线程，从而加重计算机的负担。

因此，这种调整不太靠谱，并且取决于你的应用服务器是否有一个好的流量模型。它有助于让你应用的性能百尺竿头更进一步。

10.3 EJB会话Bean

本节考察 EJB 3.0 会话 Bean 的性能。Java EE 容器管理 EJB 生命周期的方法很特殊，本节中的准则有助于确保容器管理生命周期时不会影响应用的性能。

10.3.1 调优EJB对象池

因为 EJB 对象创建（和销毁）的代价很高，所以它们通常保存在对象池中。如果没有池化，调用 EJB 包括以下步骤：

- 创建 EJB 对象
- 处理标注并且将依赖的资源注入这个新 EJB 对象
- 调用标注为 `@PostConstruct` 的方法
- 如果是状态 Bean，则调用标注为 `@Init` 的方法或者 `ejbCreate()` 方法
- 执行业务方法
- 调用任何标注为 `@PreRemove` 的方法
- 如果是状态 Bean，则调用 `remove()` 方法

如果从池中获取 EJB，则只需要调用执行业务方法——其余 6 个步骤都可以跳过。虽然通常情况下并不需要对象池（参见第 7 章），但如果初始化对象的代价高，就值得池化。

EJB 对象池化的代价

Java EE 应用服务器可以用不同大小的 EJB 池来进行测试，从而衡量从池中获取对象和因需创建对象的不同性能，所以 EJB 池可以发挥对象池的益处。

在这个例子中，我在 GlassFish 4.0 应用服务器中配置了标准的 `StockServlet`。应用中的无状态 Bean 完全没有初始化的开销。虽然有 `@PostConstruct` 方法，但是方法体是空的。

`@PostConstruct` 方法通常用于初始化资源，比如，可以执行（代价相对较高的）Java 命名和目录接口（JNDI）查找。为了模仿这种情况，我把 `StorkServlet` 的 `@PostConstruct` 方法改为 `sleep`，让它模拟时间消耗，或者执行一些初始化代码。

表 10-2 是在不同 EJB 池大小、不同 `@PostConstruct` 方法睡眠时间（模拟初始化时间）下模拟 64 个客户端访问应用时的响应时间。

表10-2：对象池对EJB响应时间的影响

EJB池大小	初始化时间（毫秒）	平均响应时间（秒）
1	0	0.37
64	0	0.37
1	25	0.40
64	25	0.37
1	50	0.42
64	50	0.37

如果初始化不需要时间，EJB 池就没什么好处。当初始化需要 25 毫秒或者 50 毫秒，并且池的大小为 1 时——意味着每次调用都会创建一个 EJB 对象——不出所料，平均响应时间拉长了。

由于这个 EJB 池中只有 64 个（小）对象，因此不太可能发生 GC。这是好的对象池的另一个关键特性：小才是好。

只有在应用服务器池中还有可用的 EJB 对象时，性能才会提高，所以必须将应用服务器中的 EJB 对象数配置成应用同时使用的 EJB 数。如果应用使用 EJB 但没有池化的实例，应用服务器就会开启 EJB 对象的完整生命周期，从创建、初始化、使用到销毁 EJB 对象。

当然，应用所依赖的对象数取决于该应用如何被使用。通常情况下，由于一个请求最多只需要一个 EJB，所以在开始的时候一般需要确保 EJB 池中的对象数和应用服务器中的工作线程数一样多。请注意，EJB 池是按类型分的：如果应用有两个 EJB 类，应用服务器就会使用两个池（每个池都可以设置线程数）。

应用服务器不同，EJB 池的调优方式也不相同，不过通常来说，每个 EJB 池都有一个全局（或默认）配置，需要不同配置的 EJB 可以覆盖该选项（通常在它们的部署描述符中）。例如，对 GlassFish 应用服务器来说，EJB 容器默认每个池中有 32 个 EJB 实例，且在 sun-ejb-jar.xml 文件的以下段落中可以配置单个 bean 池的大小：

```
<bean-pool>
  <steady-pool-size>8</steady-pool-size>
  <resize-quantity>2</resize-quantity>
  <max-pool-size>64</max-pool-size>
  <pool-idle-timeout-in-seconds>300</pool-idle-timeout-in-seconds>
</bean-pool>
```

这个例子中 EJB 池的最大值扩大了一倍，是 64。

将 EJB 池大小设置为很大值的代价通常不是非常高。池中没有使用的实例会略微降低 GC 的效率，但通常来说，池不会很大，未使用的实例不会有很明显的影响。有个例外，即如果 EJB 占用了大量内存，GC 的影响就会变大。然而，从上面的 XML 可以看出，应用服务器通常用一个池的稳定值和最大值来管理池。在上面的例子中，如果流量主要来自 EJB 中的 10 个实例（比如 10 个并发请求），一直只有 10 个 EJB 实例，那么池就永远不会达到最大值 64。

如果有短暂的流量高峰，池会创建这 64 个实例，随着流量的衰减，这些 EJB 就会空闲。一旦空闲 300 秒，就会被销毁，内存就可以被 GC。这使得池对 GC 的影响最小。

因此，要更关心 EJB 池稳定值的调优，而不是最大值的调优。

快速小结

1. EJB 池是对象池的典型范例：初始化代价高，数量相对较少，所以池化更为有效。
2. 通常来说，EJB 池的大小包括稳定值和最大值。对于特定的环境，两种值都需要调优，但从长期来看，为了降低对垃圾收集器的影响，应更注重稳定值的调优。

10.3.2 调优EJB缓存

对于状态会话 Bean，还需要考虑另外一个因素，即它们有可能被钝化（Passivation）：为了节约内存，应用服务器会选择将 Bean 的状态序列化并保存到磁盘上。这对性能会有很严重的影响，绝大多数情况下应该极力避免。

坦白说，我建议在所有情况下都避免这么做。关于钝化常见的争论是，会话空闲了几个小时或者几天，该怎么办。当用户重新回到系统时（几天后），你总希望他能找回完整的状态数据。这种情形的问题在于，它假定 EJB 会话是唯一重要的状态数据。但通常来说，EJB 与 HTTP 会话会有关联，而我们并不建议长时间保留 HTTP 会话。如果应用服务器的某种非标准特性可以将 HTTP 会话保存到磁盘，并且能配置成同时钝化 HTTP 会话和 EJB 会话（持续的时间也相同），这就有意义了。然而即便如此，其他的外部状态也可能会缺失。（比如，用户购物车中的物品失效了怎么办？）

如果需要长时间存活的状态，通常你需要绕过常规的 Java EE 状态机制。

与会话关联的状态 Bean 并没有保存在 EJB 池中，而是保存在 EJB 缓存中。因此，必须对 EJB 缓存进行调优，以便容纳应用中同时活跃的最大会话量。如果容纳不了，最近最少使用的会话将会被钝化。如前所述，不同的应用服务器实现的方式也不同。GlassFish 默认的缓存为 512，全局值可以通过域配置进行覆盖，或在 sun-ejb-jar.xml 文件中分别设置每个 EJB。

快速小结

1. EJB 缓存仅用于状态会话 Bean 与 HTTP 会话关联的时候。
2. 应该充分优化 EJB 缓存，以避免钝化。

监控 EJB 池

怎么才能知道 EJB 池和缓存的大小应该是多少？一种方法是根据应用在其预期的负载下的工作情况来进行合理的猜测。不过，想知道是否创建了太多 EJB（或钝化了太多状态会话 Bean）的唯一方法就是借助应用服务器的监控设备来进行。

图 10-1 是 GlassFish 中监控的示例。在这个例子中，EJB 累计的销毁数不为 0，表明有些 EJB 创建出来后就被销毁了，因为有些操作无法从池中获得可用的 Bean。相应地，EJB 累计的创建数大于池的最大值（这个例子中为 4）。这意味着 EJB 池过小了。

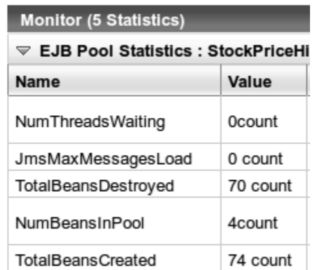


图 10-1：EJB 池监控示例

为了了解应用的性能，像这样监控统计值非常重要，但也得留意，监控本身也有代价。在这个例子中，我将 GlassFish 中的 EJB 容器监控级别设置为 `HIGH`，以便生成这些统计数据，结果总吞吐量就降低了约 5%。应用本身并没有太大影响，但对应用服务器来说，你就得注意如何进行配置和监控了。在 GlassFish 中，监控级别设为 `LOW` 的影响几乎可以忽略不计——绝大多数操作都可以进行这个级别的监控，而需要更多信息时，监控级别可以动态地设置为 `HIGH`。

10.3.3　本地和远程实例

EJB 可以通过本地或远程接口访问。在标准的 Java EE 部署中，EJB 可通过 servlet 来访问，而 servlet 可以通过本地或远程接口访问 EJB。如果 EJB 在其他系统上，则必须使用远程接口，但如果 EJB 和 servlet 在一起（这是更常见的部署方式），servlet 通常应该使用本地接口访问 EJB。

由于远程接口包含网络调用，所以上述方式看起来很合理。但这并不是主要原因——当 servlet 和远程 EJB 部署在同一个应用服务器上时，大多数服务器都足够智能，可以旁路网络调用，从而通过常规的方法调用 EJB。

优先使用本地接口的主要原因是，两类接口处理参数的方法不同。传递（或返回）给本地 EJB 的参数合乎通常的 Java 语义：原生类型通过值传递，而对象则通过引用传递。（或者，严格来说，对象句柄也仍然是通过值传递，只不过对象的引用使对象看起来是通过引用传递的。）

而传递（或者返回）给远程 EJB 的参数则总是值传递。这种通过网络的传送只有一种方式：发送方将对象序列化后以字节流的方式传输出去，而接收方则反序列化字节流后重建对象。即使服务器优化本地调用，避免了网络开销，它也不能绕过序列化 / 反序列化步骤。（大多数服务器传输对象不可变时——字符串或者原生值——都能跳过序列化的步骤，但

这不是一般情况。）无论服务器写得多好，使用远程 EJB 接口总是比本地接口慢。

Java EE 还包括其他部署场景。例如，可以将 servlet 和 EJB 部署在不同层上，且普通应用可以通过远程接口访问 EJB。也常常会有业务或者功能上的原因而使网络结构受限，例如，假设 EJB 需要访问企业数据库，你可能想将数据库放在防火墙后面的机器上，而防火墙则隔开了 servlet 容器和数据库。这些因素都是性能问题中需要重点考虑的。但严格地从性能角度来说，将访问 EJB 的组件和 EJB 部署到一起并使用本地接口总是比使用远程协议要快。

说到远程协议，所有的远程 EJB 都必须支持 IIOP（CORBA）协议。这十分有利于互通性，特别是对那些不是用 Java 编写的程序来说。对于远程访问来说，Java EE 服务器供应商也可以使用其他协议，包括专用协议。通常来说，这些专用协议都比 CORBA 快（这就是为什么供应商开发它的首要原因）。所以，如果必须使用远程 EJB 调用（不考虑不同语言之间的互通性），可以考虑那些应用服务器供应商所提供的访问协议选择。

快速小结

即便在同一个服务器中，调用 EJB 远程接口也对性能有很大的影响。

10.4　XML和JSON处理

对于部署在 Java EE 应用服务器上的 servlet 应用来说，它们的输出会在浏览器中显示，而返回给用户的数据几乎总是 HTML。本节涵盖了一些如何处理这些数据交换的最佳实践。

程序之间交换数据也可以使用应用服务器，特别是通过 HTTP。Java EE 支持多种基于 HTTP 的数据传输：成熟的 Web Service 使用 JAX-WS，RESTful 使用 JAX-RS，甚至你可以自己调用 HTTP。这些 API 的共同点是，它们都使用基于本文的数据传输（基于 XML 或 JSON）。虽然 XML 和 JSON 的数据呈现有很大的不同，但 Java 处理它们的方式是类似的，并且性能的考量点也是类似的。

这并不意味着两种呈现之间没有功能上的重要差别。一般来说，选择哪种呈现还应该依据算法和可编程性上的考虑，而不仅仅是性能。如果目的是与其他系统交互，那选择何种方式就取决于接口定义。对于复杂应用来说，处理 Java 对象通常要比遍历文档树要腰容易得多；这种情况下，JAXB（即采用 XML）是更好的选择，至少可以节省时间：Java EE 7 遵循 JSR 353（提供文档模型的标准解析）只支持 JSON-P。编写本书时，JSON-B JSR（JSON 中支持类似 JAXB 的特性）还没有得到认可（但将来可能会）。

除了上述差别之外，XML 和 JSON 还有其他重要差别。所以，本节比较两者性能的真实目的在于理解如何尽可能地获得最佳性能，而不是在特定环境下如何选择优化，无论选择的是哪种呈现。

10.4.1　数据大小

10.1 节“Web 容器的基本性能”显示了数据大小对整体性能的影响。在分布式网络环境

中，数据大小是很重要的。关于这方面，通常都认为 JSON 比 XML 小，虽然差别通常不大。在本节的测试中，我从 eBay 请求获取最畅销的 20 件商品，并用 XML 和 JSON 返回。例子中的 XML 有 23 031 字节，JSON 比较小，只有 16 078 字节。但 JSON 数据之间没有空格，所以易读性差，不过可读性并不是目标，所以并不碍事。XML 则与之不同，结构明晰，有许多空格，去掉空格后可以缩减为 20 556 字节。不过与 JSON 相比，字节数仍然有 25% 的差别，绝大部分是因为 XML 元素的结束标记。通常来说，这些结束标记总会使输出的 XML 较大。值得注意的是，有许多网站可以将 XML 自动转换成 JSON。

样本数据有效负载

贯穿本节的样本数据来自 eBay。像许多公司一样，eBay 为开发人员提供接口，以便他们在自己的应用中使用。通常来说，数据以 XML 或 JSON 的格式获取。

比如，获取 eBay 上销量排行榜前 20 位商品的列表。下面是简化后的 XML 样本数据：

```
<xml version="1.0" encoding="UTF-8"?>
  <FindPopularItemsResponse xmlns="urn:ebay:apis:eBLBaseComponents"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="urn:ebay:apis:eBLBaseComponents docs/xsd/ebay.xsd">
   <Timestamp>2013-03-29T01:57:46.530Z</Timestamp>
   <Ack>Success</Ack>
   <Build>E815_CORE_APILW2_15855352_R1</Build>
   <Version>815</Version>
   <ItemArray>
    <Item>
      <ItemID>140356481394</ItemID>
      ……其他17个属性 ……
    </Item>
    …… 其他相同结构的19个元素 ……
   </ItemArray>
 </FindPopularItemsResponse>
```

JSON 数据与此类似（实际上它没有空格，此处只是为了增加可读性）：

```
{"Timestamp":"2013-03-29T02:17:14.898Z",
 "Ack":"Success",
 "Build":"E815_CORE_APILW2_15855352_R1",
 "Version":"815",
 "ItemArray":{
    "Item":[{"ItemID":"140356481394", …… 其他17个属性 …… }],
    …… 其他相同结构的19个元素 ……
 },
}
```

无论采用哪种格式传输，数据压缩都能带来巨大的好处。实际上，两种格式压缩之后的大小非常接近：JSON 压缩后的大小为 3471 字节，XML 压缩后的大小为 3742 字节。如此一来，数据大小上的差异就不那么重要了，而传递压缩数据也和传递其他压缩后的 HTTP 数据一样，都有好处。

快速小结

和 HTML 数据一样，程序中的数据也能从减少空格和压缩中获得巨大的益处。

10.4.2　解析和编组概述

给定一组 XML 或 JSON 字符串，程序必须将其转换成适合 Java 处理的数据。依据程序的上下文和输出结果，这个过程被称为编组（marshal）或解析。反过来——从数据生成 XML 或 JSON 串——则被称为解组（unmarshal）。

一般来说，处理这些数据涉及以下四种技术。

标识符解析器（Token parser）

　解析器遍查输入数据中的标识符，当发现标识符时则回调相应对象上的方法。

拉模式解析器（Pull parser）

　输入的数据与解析器关联，程序从解析器中请求（或拉取）标识符。

文档模型（Document model）

　输入数据被转换成文档风格的对象，以便程序在查找数据片段时可以遍历。

对象呈现（Object representation）

　通过与输入数据对应的预定义类，可以将数据转换成一个或多个 Java 对象（例如，可以用预定义的 `Person` 类来转换关于人的数据）。

虽然上述技术大体上按照性能从慢到快的顺序排列，但它们之间最主要的差别是功能而不是性能。前两种技术在功能上没有很大差别：它们都适用于大多数只需扫描一次就能提取信息的算法。不过解析器所能做的只是简单的扫描。解析器模式并不非常适合那种需要随机访问的数据，或者需多次遍历的数据。为了应对这些情况，使用简单解析器的程序应该构建内部的数据结构，虽然这只是个简单的编程问题，但文档对象模型和 Java 对象模型已经提供了结构化的数据，可能比你自己定义新结构要容易。

实际上，这就是使用解析器和数据编组之间的真实差别。前面两种是纯粹的解析器模式，取决于如何用解析器提供的方式处理数据逻辑。下面两项是数据编组器：它们必须使用解析器处理数据，但它们所提供的数据呈现可以用在更复杂的程序逻辑中。

所以，采用何种技术首先取决于应用是什么样的。如果程序只需要简单地过一遍数据，那么简单地使用最快的解析器是最有效的。如果数据需要保存为应用所定义的简单结构，那么直接使用解析器也是合适的；例如，示例数据中商品条目的价格需要保存为 `ArrayList`，以便应用的其他部分进行处理。

数据格式为重的时候，使用文档模型更合适。如果必须保留数据格式，那文档的格式转换就会很容易：数据读入后转成文档格式，用某种方法更改，然后可以很容易地写到新的数据流中。

为了尽可能便利，对象模型提供 Java 语言层面的数据呈现。数据可以通过对象及其属性的方式来操作。数据编组时所增加的复杂性（绝大部分）对开发人员透明，并且会使应用略微慢一些，但开发人员代码生产率上的提高可以抵消这个问题。

本节的示例读取有 20 个条目的 XML 或者 JSON 文档，并将条目 ID 保存到 `ArrayList` 中。对某些测试来说，只需要前 10 个条目。这是为了模拟真实世界里经常发生的事，即返回的数据总是超过实际所需要的数据。在 Web 服务的设计考量中，这是很好的点：调用的建立需要一些时间，用更少的远程调用（即便需要大量数据）而不是大量小的远程调用。

尽管所有的示例都展示了这类常见操作，但关键点并不在于直接比较这部分任务的性能。而是说，每个示例展示的是如何在所选择的框架下最有效地执行操作，因为框架的选择并不单是考虑解析和数据编组的性能。

快速小结

1. Java EE 应用中有很多办法处理程序所需要的数据。
2. 虽然这些技术给开发人员提供了很多功能，但数据处理本身的代价也增加了。不要因此影响你在应用中选择正确处理数据的方法。

10.4.3 选择解析器

编程中所用的所有数据都必须能被解析。对应用来说，选择直接使用解析器，还是通过序列化框架间接使用解析器，对于数据操作的整体性能至关重要。

1. 拉模式解析器

从开发者的角度来看，拉模式的解析器最容易使用。在 XML 的世界中，广为人知的拉模式解析器就是 StAX（Streaming API for XML）解析器。JSON-P 只提供拉模式解析器。

拉模式解析器依据需要从流中获取数据。本节测试所用的基本拉模式解析器的主逻辑就是下面的这个循环：

```
XMLStreamReader reader = staxFactory.createXMLStreamReader(ins);
while (reader.hasNext()) {
    reader.next();
    int state = reader.getEventType();
    switch (state) {
        case XMLStreamConstants.START_ELEMENT:
            String s = reader.getLocalName();
            if (ITEM_ID.equals(s)) {
                isItemID = true;
            }
            break;
        case XMLStreamConstants.CHARACTERS:
            if (isItemID) {
                String id = reader.getText();
                isItemID = false;
                if (addItemId(id)) {
                    return;
                }
            }
```

```
                break;
            default:
                break;
        }
    }
```

解析器返回一组 token。这个例子中的大多数 token 都会被丢弃。当遇到起始类型的 token 时，就会检查是不是 `ITEM_ID`。如果是，则下一个字符 token 就是应用所需要保存的 ID。ID 可通过 `addItemId()` 保存，如果成功保存 ID 则返回 `true`。一旦发生，循环就会返回，不再处理输入流中剩下的数据。

从概念上说，JSON 解析器的工作方式与此一模一样，只是有一些 API 调用上的变化：

```
while (parser.hasNext()) {
    Event event = parser.next();
    switch (event) {
        case KEY_NAME:
            String s = parser.getString();
            if (ITEM_ID.equals(s)) {
                isItemID = true;
            }
            break;
        case VALUE_STRING:
            if (isItemID) {
                if (addItemId(parser.getString())) {
                    return;
                }
                isItemID = false;
            }
            continue;
        default:
            continue;
    }
}
```

只处理必要的数据可以给性能带来可预见的好处。表 10-3 列出了解析样本文档的平均时间（毫秒），假设条件从解析 10 个条目后即可退出循环，到处理整个文档。解析 10 个条目后退出并没有节约 50% 的时间（因为文档的其他段落也需要解析），但差别还是很显著的。

表10-3：拉模式解析器的性能

处理的条目数	XML解析器（毫秒）	JSON解析器（毫秒）
10	143	68
20	265	146

2. 推模式解析器

标准的 XML 解析器是 SAX（Simple API for XML）解析器。SAX 解析器是一种推模式解析器：读入数据，当发现 token 时，就会执行类中处理该 token 的回调方法。下面测试中的解析逻辑与之前相同，不过现在逻辑放在了类所定义的回调方法中：

```
protected class CustomizedInnerHandler extends DefaultHandler {
    public void startElement(String space, String name,
```

```
                      String raw, Attributes atts) {
                 if (name.length() == 0)
                     name = raw;
                 if (name.equalsIgnoreCase(ITEM_ID))
                     isItemID = true;
        }

        public void characters(char[] ch, int start,
                     int length) throws SAXDoneException {
            if (isItemID) {
                String s = new String(ch, start, length);
                isItemID = false;
                if (addItemId(s)) {
                    throw new SAXDoneException("Done");
                }
            }
        }
    }
```

这里程序逻辑上唯一的差别是，必须以抛出异常的方式来通知解析完成了，因为这是 XML 推模式解析框架检测到解析应该停止的唯一方法。这个例子中，应用所抛出的异常是 SAXDoneException。一般来说，任何 SAXException 都可以抛出，这个例子中使用的是该异常的子类，使得程序其他部分的逻辑可以区分哪个是实际错误，哪个是通知解析终止的信号。

SAX 解析器比 StAX 快，虽然性能上的差别很小——选择哪种解析器应该取决于开发中哪种模型更容易。表 10-4 展示了推模式解析器和拉模式解析器在处理时间上的差异。

表10-4：推模式解析器的性能

条目数	XML StAX解析器（毫秒）	XML SAX解析器（毫秒）
10	143	132
20	265	231

JSON-P 没有相应的推模式解析器模型。

3. 其他解析机制的实现和解析器工厂

XML 和 JSON 规范定义了解析器的标准接口。JDK 提供了 XML 解析器的参考实现，JSON-P 项目则提供了 JSON 解析器的参考实现。应用可以使用任意解析器（当然，只要该解析器实现了所需要的接口）。

解析器是从解析器工厂中获得的。将解析器工厂设置成返回所需解析器的实例（而不是默认解析器），就可以使用不同的解析器实现。这其中隐含着某些性能问题。

- 工厂初始化的代价昂贵：确保可以通过全局（或至少是线程本地变量）引用的方式重用工厂。
- 工厂可通过多种不同的方式进行配置，其中一些配置（包括默认配置）从性能的角度来看并不是最优的。
- 其他的解析器实现可能比默认的更快。

> ### 工厂和解析器的重用
>
> XML 和 JSON 解析器工厂的创建代价很高。幸运的是，工厂是线程安全的，所以很容易保存在全局静态变量中，可以在需要的时候重用。
>
> 但一般来说，解析器无法重用，也不是线程安全的。因此，解析器通常是因需而建。
>
> SAX 解析器的一个优点是可以重用解析器对象。重用时，只需要在使用解析器之前调用 `reset()` 方法即可。不过解析器仍然不是线程安全的，所以务必确保同一时间只在单个线程中重用解析器。

让我们依次来看上述几点。

为求平均的解析速度，这些测试解析了 1 百万次数据（10 000 次预热解析之后）。下面的示例代码确保只构造一次工厂，在测试开始时调用的初始化方法中完成。每轮测试中的解析器实例则由工厂因需而创建。由此，SAX 测试包含的代码如下所示：

```
SAXParserFactory spf;
// 只在程序初始化时调用一次
protected void engineInit(RunParams rp) throws IOException {
    spf = SAXParserFactory.newInstance();
}
// 每轮迭代时调用
protected XMLReader getReader() Throws SAXException {
    return spf.newSAXParser().getXMLReader();
}
```

StAX 解析器与此类似，调用 `XMLFactory.newInstance()` 获得工厂（类型为 `XMLInputFactory`），然后调用工厂的 `createStreamReader()` 方法获得 `StreamReader`。对于 JSON，相应的调用方法是 `Json.createParserFactory()` 和 `createParser()`。

如果要用另一种解析器实现，我们必须用另一种工厂，才能使工厂的调用返回所需要的实现。这就是关于工厂配置的第二点：确保所用的工厂是经过优化设定的。

可以通过 3 种方法设定 XML 工厂。此处所用的工厂（`javax.xml.stream.XMLInputFactory`）默认设定的是 StAX 解析器。为了覆盖默认的 SAX 解析器，需设置成 `javax.xml.parsers.SAXParserFactory`。

为确定使用的是哪种工厂，需要按以下顺序查找选项。

1. 使用由系统属性 `-Djavax.xml.stream.XMLInput Factory=my.factory.class` 指定的工厂。
2. JAVA/jre/lib 下的文件 jaxp.properties 内所指定的工厂，类似这样一行：

   ```
   javax.xml.stream.XMLInputFactory=my.factory.class
   ```

3. 在 classpath 上搜索文件 META-INF/services/javax.xml.stream.XMLInputFactory。该文件需要包含单独的一行 `my.factory.class`。
4. 使用 JDK 定义的默认工厂。

上面第 3 种方法有明显的性能问题，特别是在环境设置了很长的 classpath 的时候。为了查

看某个备选的实现是否已被设定，必须扫描整个 classpath，搜索每个入口中的 META-INF/services 目录下的特定文件。而且，每次创建工厂时都会重复这个查找过程。所以，如果类加载器没有缓存资源的查找结果（大多数类加载器没有缓存），初始化工厂的代价就非常高。

更好的做法是用前两种办法配置应用。系统会依上述列表的顺序查找工厂，一旦找到，搜索过程就会停止。

这两种方法的不足之处在于，它们是全局的，会影响应用服务器上的所有代码。如果两个不同的企业应用部署到了同一个服务器上，并且需要不同的解析器工厂，那服务器就要必须依靠在 classpath 上搜索工厂的技术了（可能会很慢）。

发现解析器工厂的方法甚至还影响了默认工厂：JDK 必须要搜索完 classpath 后才知道使用默认工厂。因此，即便你使用默认工厂，你也应该通过配置全局系统属性或 Java 运行时环境（JRE）属性文件来指向默认实现。否则，只有在第 3 步花费了昂贵代价搜索之后，才会使用默认工厂（列表中的第 4 项）。

对 JSON 来说，配置有少许不同：指定其他实现的唯一办法是，在 META-INF/services 下指定一个名为 javax.json.spi.JsonProvider 的包含新 JSON 解析器实现类的类名的文件。不幸的是，查找 JSON 工厂时，没有办法避免搜索整个 classpath。

选择解析器的最后一个性能考量点是备选实现的性能。本节只是对一些解析器实现性能的快速浏览，没有必要在意表面上的结果。不同的实现之间总有差异。就性能而言，不同实现之间可能各有千秋。某些时候，备选实现会比参考实现快（到 JDK 新的发布版或者新的 JSON-P 参考实现时，参考实现可能就会超过备选实现）。

比如说，在编写本书时，Woodstox 的 StAX 解析器（http://woodstox.codehaus.org/）就比 JDK 7 和 8 所带的解析器要快一些（见表 10-5）。

表10-5：StAX解析器的性能

数据条数	JDK StAX解析器（毫秒）	Woodstox StAX解析器（毫秒）
10	143	125
20	265	237

而 JSON 解析器的状况要混乱得多。在编写本书时，JSON-P 规范的最终稿刚刚制定好，但还没有 JSR 353 兼容的 JSON 解析器实现。对于其他 JSON 解析器来说，最后遵循 JSR 353 API 只是一个时间问题。

具体应视情况而定，所以查找其他 JSON 实现，看看它们是否有更好的性能不失为一个好主意。一种实现是 Jackson JSON 处理器（目前不兼容 JSR 353），它已经实现了基本的拉模式解析器（准确来说，并不是 JSR 353 的 API 调用）。参见表 10-6。

表10-6：JSON解析器的性能

数据条数	Java EE JSON解析器（毫秒）	Jackson JSON解析器（毫秒）
10	68	40
20	146	74

新的 JSR 参考实现通常也是这样，就像 JDK 7 XML 解析器比前一个版本快很多一样，新的 JSON-P 解析器的性能预计也会有巨大的提升。（实际上，本节测试所用的 JSON 解析器版本为 1.0.2，比初始的 1.0 版本快约 65%。）

快速小结

1. 选择的解析器是否合适，对应用的性能有巨大的影响。
2. 推模式解析器通常比拉模式解析器快。
3. 查找解析器工厂的算法非常耗时；如果可能的话，应该通过系统属性直接指定工厂而不是用现有的实现。
4. 在不同的时间点上，最快的解析器实现的赢家可能会不同。适当的时候，应该从备选的解析器中找。

10.4.4 XML验证

解析器可依据一个 schema（意为“模式”）对 XML 数据进行验证，拒绝语法不正确的文档——指缺少某些必要的信息，或者包含了不该有的信息的文档。此处所说的“语法正确”是指文档内容，如果文档有语法错误（比如文档没有包含在 XML 标签中，或者缺少 XML 闭合标签等），所有解析器都不会接收该文档。

这种验证是 XML 相比 JSON 所具有的一个优点。解析 JSON 文档时你可以自己提供验证逻辑，但解析 XML 时，解析器能替你做这些验证。但这个好处是有性能代价的。

XML 验证是依据一个或多个 schema 或 DTD 文件进行的。虽然 DTD 的验证更快，但 XML schema 更灵活，现在是 XML 世界中的主流。schema 比 DTD 慢的一个原因是，schema 通常在多个文件中设定。所以减少验证成本的第一个方法就是整合 schema 文件：schema 文件越多，验证的代价越高。需要在多个文件的可维护性和性能收益之间进行权衡。不幸的是，由于 schema 文件维护了不同的命名空间，所以整合起来并不容易（就像 CSS 或 JavaScript 文件那样）。

从何处装载 schema 文件对性能有极大的影响。如果必须反复从网络上装载 schema 或 DTD，性能就会变糟糕。理想情况下，schema 文件应该随着应用代码一起分发，这样就能从本地文件系统装载了。

对常见的 SAX 验证来说，只需要用代码为 SAX 解析器工厂设置一些属性即可（这只对 SAX 解析器有效；对 StAX 解析器而言，除非使用本节后面讨论的 `Validator` 对象，否则验证依据的是 DTD 文件而不是 schema）。

```
SAXParserFactory spf = SAXParserFactory.newInstance();
spf.setValidating(true);
spf.setNamepsaceAware(true);
SAXParser parser = spf.newSAXParser();
// 注意:创建解析器时可以执行上面的几行代码
// 如果重用该解析器,而不是调用parser.reset(),则需要设置属性
parser.setProperty(JAXPConstants.JAXP_SCHEMA_LANGUAGE,
        XMLConstants.W3C_XML_SCHEMA_NS_URI);
XMLReader xr = parser.getXMLReader();
xr.setErrorHandler(new MyCustomErrorHandler());
```

解析器默认没有验证，所以得先调用 `setValidating()`，然后设置属性，将验证所依据的语言告诉解析器——本例中为 W3C XML schema 语言（例如 XSD 文件）。最后，设置解析器在验证出错时的处理程序。

这种处理方式——XML 文档的默认处理方式——会在每次解析新文档时重读 schema，即便解析器本身已经重用。为了更好的性能，可以考虑重用 schema。

即便从文件系统装载 schema，重用 schema 也会有很大的好处。装载 schema 时，必须解析和处理它（毕竟它自己也是 XML 文档）。保留处理结果并重用可以极大地提升 XML 的处理效率。这在绝大多数应用场景下都是正确的：应用接收和处理成千上万的 XML 文档，而所有这些文档都遵循相同的一个（或一组）schema。

重用 schema 的方法有两种，第一种（只对 SAX 解析器有效）是创建 schema 对象并与 `SAXParserFactory` 关联：

```
SchemaFactory sf = SchemaFactory.newInstance(
        XMLConstants.W3C_XML_SCHEMA_NS_URI);
StreamSource ss = new StreamSource(rp.getSchemaFileName());
Schema schema = sf.newSchema(new Source[]{ss});
SAXParserFactory spf = SAXParserFactory.newInstance();
spf.setValidating(false);
spf.setNamespaceAware(true);
spf.setSchema(schema);
parser = spf.newSAXParser();
```

请注意，这个例子中调用 `setValidating()` 时参数为 `false`。`setSchema()` 和 `setValidating()` 是两种互相矛盾的文档验证方法[1]。

重用 schema 对象的第二种方法是使用 `Validator`，将解析与验证分离，使得两种操作可以在不同时间进行。用 StAX 解析器解析时，可以在流的验证过程中嵌入特定的 reader 来进行验证。

使用 `Validator` 时，首先要创建特定的 reader。reader 的处理逻辑和之前相同：查找起始元素（start element）`itemID`，找到之后保存这些 ID。不过，这些操作必须委托给默认的 StAX 流 reader：

```
private class MyXMLStreamReader extends StreamReaderDelegate() {
    XMLStreamReader reader;
    public MyXMLStreamReader(XMLStreamReader xsr) {
        reader = xsr;
    }

    public int next() throws XMLStreamException {
        int state = super.next();
        switch (state) {
            case XMLStreamConstants.START_ELEMENT:
                ……处理起始元素Item ID……
                break;
            case XMLStreamConstants.CHARACTERS:
```

注 1：因为 `setValidating()` 只包括 DTD 的验证。——译者注

```
                    ……如果是item id,则保存当前的字符。
                    break;
            }
            return state;
        }
    }
```

接下来，将这个 reader 与 `Validator` 所用的输入流关联。

```
    SchemaFactory sf = SchemaFactory.newInstance(
            XMLConstants.W3C_XML_SCHEMA_NS_URI);
    StreamSource ss = new StreamSource(rp.getSchemaFileName());
    Schema schema = sf.newSchema(new Source[]{ss});
    XMLInputFactory staxFactory = XMLInputFactory.newInstance();
    staxFactory.setProperty (XMLInputFactory.IS_VALIDATING, Boolean.FALSE);
    XMLStreamReader xsr = staxFactory.createXMLStreamReader(ins);
    XMLStreamReader reader = new MyXMLStreamReader(xsr);
    Validator validator = schema.newValidator();
    validator.validate(new StAXSource(new StaxSource(reader)));
```

`validate()` 在进行常规验证时还会调用 `StreamReaderDelegate`（此处为 `MyXMLStreamReader`），它会从输入数据中解析所需要的信息（实质上是验证过程的意外收获）。

这种方法的不足之处在于，处理过程无法在读取若干条数据之后干净利索地终止。`next()` 依旧可以在抛出异常之后捕获该异常——就像前面的 `SAXDoneException`。此处有个问题，即默认的 schema 监听器在处理遇到异常时会打印错误信息。

表 10-7 列出了所有这些操作的影响。与简单的解析（不进行验证的）相比，带有默认验证的解析会付出更大的性能代价。重用 schema 可以弥补一些性能损耗，并能确保 XML 文档是合乎语法规范的，但验证总会有些显著的代价。

表10-7：XML文档验证的性能

解析模式	SAX（毫秒）	StAX（毫秒）
无验证	231	265
默认验证	730	N/A
重用 schema 的验证	649	1392

快速小结

1. 如果业务需要进行 schema 验证，那就用它。只是要留意，验证对解析数据的性能会带来显著的损耗。
2. 总是重用 schema，以将验证对性能的影响降至最低。

10.4.5 文档模型

构建文档对象模型（Document Object Model，DOM）或 JSON 对象只需要一组相对简单的方法调用。对象本身是随着底层的解析器而创建的，所以优化性能的一个重要举措是配置好解析器（DOM 默认使用 StAX 解析器）。

DOM 对象是随着 `DocumentBuilder` 对象而创建的，而 `DocumentBuilder` 来自 `Document-`

BuilderFactory。默认的 DocumentBuilderFactory 可通过属性 javax.xml.parsers.DocumentBuilderFactory（或者文件 META-INF/services/javax.xml.parsers.DocumentBuilderFactory）来指定。由于创建解析器时所配置的属性对性能优化很重要，所以创建 DocumentBuilder 时配置的系统属性也就变得很重要了。

和 SAX 解析器一样，只要在使用前调用 reset() 方法就能重用 DocumentBuilder 对象。

JSON 对象是随着 JsonReader 对象而创建的，而 JsonReader 可直接来自 Json 对象（通过调用 Json.createReader()）或来自 JsonReaderFactory 对象（通过调用 Json.createJsonReaderFactory()）。虽然 JSR 353 RI 现在还没有支持任何配置选项，但通过属性 Map 可以配置 JsonReaderFactory。JsonReader 对象不能重用。

如表 10-8 所示，与简单解析数据相比，DOM 的创建代价比较高。

表10-8：DOM与JSON解析的性能对比

操作	XML（毫秒）	JSON（毫秒）
解析数据	265	146
构建文档	348	197

构建文档的时间包括解析的时间加上创建文档对象结构的时间——所以从这张表可以推算出，对 XML 而言构建文档结构的时间大约占了总时间的 33%，对 JSON 而言则占了 25%。对更复杂的文档来说，构建文档模型所占用的时间百分比会更大。

之前的解析测试有时候可能只关心前 10 个条目。如果对象展现类似地也包括这前 10 个元素，那就有两种选择。首先创建对象，调用各种方法遍历对象，并丢弃不需要的条目。这是 JSON 对象唯一的选择。

DOM 对象可以用 DOM 级别 3 的属性建立过滤解析器。首先要创建一个解析过滤器：

```
private class InputFilter implements LSParserFilter {
    private boolean done = false;
    private boolean itemCountReached;
    public short acceptNode(Node node) {
        if (itemCountReached) {
            String s = node.getNodeName();
            if ("ItemArray".equals(s)) {
               return NodeFilter.FILTER_ACCEPT;
            }
            if (done) {
               return NodeFilter.FILTER_SKIP;
            }
            // 我们不需要元素</Item>
            if ("Item".equals(s)) {
                done = true;
            }
        }
        return NodeFilter.FILTER_ACCEPT;
    }

    public int getWhatToShow() {
```

```
            return NodeFilter.SHOW_ALL;
        }

        public short startElement(Element element) {
            if (itemCountReached) {
                return NodeFilter.FILTER_ACCEPT;
            }
            String s = element.getTagName();
            if (ITEM_ID.equals(element.getTagName())) {
                if (addItemId(element.getNodeValue())) {
                    itemCountReached = true;
                }
            }
            return NodeFilter.FILTER_ACCEPT;
        }
    }
```

每个元素会调用两次解析过滤器：解析元素开始时，调用 `startElement()`，结束元素结束时，调用 `acceptNode()`。如果元素不应在最终的 DOM 文档中出现，就应在上述两个方法之一中返回 `fiLTER_SKIP`。在这个例子中，`startElement()` 用来追踪有多少条目已经被处理了，而 `acceptNode()` 则用来判定是否整个元素应该被跳过。请注意，`acceptNode()` 中也需要追踪结尾的 `</Item>` 标签，以免跳过。也请注意，只有类型 `ItemArray` 的子节点元素才会被跳过，XML 文档的其他元素则不应该被跳过。

为了设定输入过滤器，可以使用以下代码：

```
    System.setProperty(DOMImplementationRegistry.PROPERTY,
          "com.sun.org.apache.xerces.internal.dom.DOMImplementationSourceImpl");
    DOMImplementationRegistry registry =
          DOMImplementationRegistry.newInstance();
    DOMImplementation domImpl = registry.getDOMImplementation("LS 3.0");
    domLS = (DOMImplementationLS) domImpl;
    LSParser lsp = domLS.createLSParser(DOMImplementationLS.MODE_SYNCHRONOUS,
                       "http://www.w3.org/2001/XMLSchema");
    lsp.setFilter(new InputFilter());
    LSInput lsi = domLS.createLSInput();
    lsi.setByteStream(is);
    Document doc = lsp.parse(lsi);
```

最后创建 `Document` 对象，如同它没有过滤输入——不过在这个例子中，结果文档要小得多。这是过滤的关键点：在实际的解析和过滤过程中，生成过滤的文档所花费的时间比生成包含所有原始信息的文档花费的时间要多。因为文档占用的内存更小，且能减少对垃圾收集器的压力，所以这在文档需要长时间存活（或者有许多这样的文档在使用）时很有用。

表 10-9 显示的是构造只有一半（10 个）条目的 DOM 对象时，解析一般的 XML 文件的速度差异。

表10-9：过滤DOM文档所造成的影响

	标准DOM	过滤DOM
创建 DOM 所用时间	348 毫秒	417 毫秒
DOM 的大小	101 440 字节	58 824 字节

快速小结

1. 使用 DOM 和 `JsonObject` 比用简单解析器要强大得多，但构造模型所花的时间长度会很显著。
2. 过滤模型数据比构造默认模型要花费更多的时间，但对于长时间运行或者很大的文档来说，仍然是值得的。

10.4.6 Java对象模型

处理文本数据的最后一种选择是在解析相关的数据之后创建一组 Java 类实例。JSR 有此类的 JSON 建议（proposal），但没有标准。对 XML 来说，这是通过 JAXB 来实现的。

JAXB 底层用的是 StAX 解析器，所以为你的平台选择最佳的 StAX 解析器配置有助于提高 JAXB 的性能。JAXB 通过创建 `Unmarshaller` 对象来创建 Java 对象：

```
JAXBContext jc = JAXBContext.newInstance("net.sdo.jaxb");
Unmarshaller u = jc.createUnmarshaller();
```

创建 `JAXBContext` 的代价比较昂贵。幸运的是，它是线程安全的：可以创建单个全局的 `JAXBContext`，然后重用（在多个线程间共享）。但 `Unmarshaller` 对象不是线程安全的，所以每个线程必须创建一个新对象。不过 `Unmarshaller` 对象可以重用，所以将它保存在线程本地变量中（或者对象池中），将有助于提高处理大量文档时的性能。

通过 JAXB 创建对象的代价要比解析或创建 DOM 文档的代价昂贵。但权衡下来，使用这些对象还是要比遍历文档快得多（甚至，使用对象就仅仅是写常规的 Java 代码，而不是用错综复杂的 API 来访问）。此外，依据 JAXB 文档编写 XML 要比直接从文档编写 XML 快得多。表 10-10 显示了性能差异，示例文档有 20 个条目。

表10-10：JAXB编组和解组的性能

	编组（毫秒）	解组（毫秒）
DOM	348	298
JAXB	414	232

过滤 XML 和 JAX-WS

即便使用 JAXB，你需要处理的通常也只是部分 XML 数据。而一般来说，JAX-WS 会基于 JAXB 将整个 XML 转换成 Java 对象。从易用角度来看，这种方法很好，它使得应用代码更容易编写和维护。但是，如果只需要访问部分 XML，用 JAXB 处理整个文档就太奢侈了，并且所有这些 JAXB 对象会消耗太多堆内存。

在这个例子中，XML 数据应该作为 SOAP 消息的附件发送（MIME 类型为 application/xml）。附件不会被转换成 JAXB 对象，你可以用 DOM builder 过滤，或者用简单的 StAX 解析器只处理你所关心的文档部分。

快速小结

1. JAXB 将 XML 文档生成 Java 对象，以最简单的编程模型访问和使用数据。
2. 创建 JAXB 对象的代价比创建 DOM 对象的昂贵。
3. JAXB 写 XML 数据的速度要快于 DOM 对象。

10.5　对象序列化

不同系统间的数据交换可以使用 XML、JSON 和其他基于文本的格式。Java 进程间交换数据，通常就是发送序列化后的对象状态。尽管序列化在 Java 中随处可见，但 Java EE 中还有两点需要重点考虑。

- Java EE 服务器间的 EJB 调用——远程 EJB 调用——通过序列化交换数据。
- HTTP 会话状态通过对象序列化的方式来保存，这让 HTTP 会话可以高可用。

JDK 提供了默认的序列化对象机制，以实现 `Serializable` 或 `Externalizable` 接口。实际上，默认序列化的性能还有提升的空间，但此时进行过早的优化的确不太明智。特定的序列化和反序列化代码需要很多时间编写，而且也比默认的序列化代码更难维护。编写正确的序列化代码会有一些棘手，试图优化代码也会增加出错的风险。

10.5.1　transient字段

一般来说，序列化的数据越少，改进性能所需的代价就越少。将字段标为 `transient`，默认就不会序列化了。类可以提供特定的 `writeObject()` 和 `readObject()` 以处理这些数据。如果不需要这些数据，简单地将它标记为 `transient` 就足够了。

10.5.2　覆盖默认的序列化

`writeObject()` 和 `readObject()` 可以全面控制数据的序列化。正所谓“权力越多，责任越大”：序列化很容易出错。

为了了解序列化优化的困难性，以一个表示位置的简单对象 `Point` 为例：

```
public class Point implements Serializable {
    private int x;
    private int y;
    ...
}
```

在我的机器上，100 000 个这样的对象可以在 133 毫秒内序列化，在 741 毫秒内反序列化。但即便像这么简单的对象，性能——即便非常困难——也能改善。

```
public class Point implements Serializable {
    private transient int x;
    private transient int y;
    ....
    private void writeObject(ObjectOutputStream oos) throws IOException {
        oos.defaultWriteObject();
```

```
            oos.writeInt(x);
            oos.writeInt(y);
        }
        private void readObject(ObjectInputStream ois)
                                throws IOException, ClassNotFoundException {
            ois.defaultReadObject();
            x = ois.readInt();
            y = ois.readInt();
        }
    }
```

在我机器上序列化 100 000 个这样的对象仍然要花费 132 毫秒，但反序列化只需要 468 毫秒——改善了 30%。如果简单对象的反序列化占用了相当大一部分程序运行的时间，像这样优化就比较有意义。然而请当心，这会使得代码难以维护，因为字段被添加、移除了，等等。

到目前为止，代码更为复杂了，但功能上依然正确（且更快）。注意，将此技术应用到一般场景时务必要谨慎：

```
    public class TripHistory implements Serializable {
        private transient Point[] airportsVisited;
        ....
        // 注意,这段代码不正确!
        private void writeObject(ObjectOutputStream oos) throws IOException {
            oos.defaultWriteObject();
            oos.writeInt(airportsVisited.length);
            for (int i = 0; i < airportsVisited.length; i++) {
                oos.writeInt(airportsVisited[i].getX());
                oos.writeInt(airportsVisited[i].getY());
            }
        }

        private void readObject(ObjectInputStream ois)
                                throws IOException, ClassNotFoundException {
            ois.defaultReadObject();
            int length = ois.readInt();
            airportsVisited = new Point[length];
            for (int i = 0; i < length; i++) {
                airportsVisited[i] = new Point(ois.readInt(), ois.readInt();
            }
        }
    }
```

此处的字段 airportsVisited 是表示我出发或到达的所有机场的数组，按照我离开或到达它们的顺序排列。有些机场，像 JFK，在数组中出现得比较频繁，SYD（目前）只出现过一次。

由于序列化对象引用的代价比较昂贵，所以上述代码要比默认的数组序列化机制快：在我的机器上，100 000 个 Point 对象的数组序列化用时 4.7 秒，反序列化用时 6.9 秒。上述“优化”使得序列化只用了 2 秒，反序列化只用了 1.7 秒。

然而这段代码是不正确的。指定 JFK 位置的数组引用都指向相同的对象。这意味着，如果

发现数据不正确而更改单个 JFK，那数组中的所有引用都会受到影响（因为它们引用的是相同的对象）。

用上述代码反序列化数组时，这些 JFK 引用就会变为独立的、不同的对象。当某个对象更改时，就只有它发生改变，结果它的数据就不同于其他那些表示 JFK 的对象了。

这条原则非常重要，应该铭记于心，因为序列化的调优常常就是如何对对象的引用进行特殊处理。做对了，序列化的性能可以获得极大提升；做错了，就会引入不易察觉的 bug。

鉴于此，我们来考察一下 StockPriceHistory 的序列化，看看如何优化序列化。以下是这个类的字段：

```
public class StockPriceHistoryImpl implements StockPriceHistory {
    private String symbol;
    protected SortedMap<Date, StockPrice> prices = new TreeMap<>();
    protected Date firstDate;
    protected Date lastDate;
    protected boolean needsCalc = true;
    protected BigDecimal highPrice;
    protected BigDecimal lowPrice;
    protected BigDecimal averagePrice;
    protected BigDecimal stdDev;
    private Map<BigDecimal, ArrayList<Date>> histogram;
    ....
    public StockPriceHistoryImpl(String s, Date firstDate, Date lastDate) {
        prices = ....
    }
}
```

当以给定标志 s 构造 StockPriceHistoryImpl 对象时，会创建和存储 SortedMap 类型的变量 prices，键值为 start 和 end 之间的所有股票价格的时间。构造函数也设置保存了 firstDate 和 lastDate。除此之外，构造函数没有设置任何其他字段，它们都是延迟初始化。当调用这些字段的 getter 方法时，getter 会检查 needsCalc 是否为真。如果为真，就会立即计算这些字段的值。

计算包括创建 histogram，它记录了该股票特定的收盘价出现在哪些天。histogram 包含的 BigDecimal 和 Date 对象的数据与 prices 中的相同，只是看待数据的方式不同。

所有的延迟加载字段都可以由 prices 数组计算得来，所以它们都可以标记为 transient，并且在序列化和反序列化时不需要为它们做额外的工作。这个例子比较简单，因为代码已经完成了字段的延迟初始化，因此在接收数据时，可以一直延迟初始化。即便字段要即刻初始化，也仍然可以将可计算字段标记为 transient，而在 readObject() 方法中重新计算它们的值。

注意，上述做法也维护了 prices 和 histogram 对象之间的关系：重新计算 histogram 时，会将已存在的对象塞到新的 map 中。

这种做法在绝大多数情况下都能收到优化效果，但有时也会降低性能。表 10-11 就是这种情况，该表显示了 histogram 对象有无 transient 字段时进行序列化和反序列化所花费的时间，以及序列化数据的大小。

表10-11：序列化和反序列化对象（关于transient字段）所用的时间

对象	序列化时间（秒）	反序列化时间（秒）	数据大小（字节）
没有 transient 字段	12.8	11.9	46 969
histogram 为 transient	11.5	10.1	40 910

目前来看，这个例子中的对象序列化和反序列化节约了大约 15% 的时间。但这个测试实际上没有在接收时重建 histogram 对象：对象只有在接收数据的代码首次对其进行访问时才会创建。

有些时候并不需要 histogram 对象：客户端可能只关心特定日子里的股价，而不是整个 histogram。还有一些不常见的情况，比如如果总是需要 histogram，且测试中计算所有的 histogram 用时超过了 3.1 秒，那么延迟初始化字段就确实会导致性能下降。

在这个例子中，计算 histogram 并不属于这种情况——这是一种非常快的操作。一般来说，重新计算数据片段的代价很少会高于序列化和反序列化数据。但在代码优化时仍然需要考虑。

这个测试实际上并不向系统外传播数据，只是在预先分配的字节数组中写数据和读数据，所以它只是衡量了序列化和反序列化所用的时间。另外，histogram 字段标为 transient 也减少了 13% 的数据大小。通过网络传送数据时，这就变得非常重要了。

10.5.3 压缩序列化数据

上述两种方法引出了改善序列化代码性能的第 3 种方法：数据序列化之后再进行压缩，使得它可以更快地在慢速网络上传输。StockPriceHistoryCompress 在序列化时对 prices 进行了压缩：

```
public class StockPriceHistoryCompress
        implements StockPriceHistory, Serializable {

    private byte[] zippedPrices;
    private transient SortedMap<Date, StockPrice> prices;

    private void writeObject(ObjectOutputStream out)
                throws IOException {
        if (zippedPrices == null) {
            makeZippedPrices();
        }
        out.defaultWriteObject();
    }

    private void readObject(ObjectInputStream in)
                throws IOException,  ClassNotFoundException {
        in.defaultReadObject();
        unzipPrices();
    }

    protected void makeZippedPrices() throws IOException {
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        GZIPOutputStream zip = new GZIPOutputStream(baos);
        ObjectOutputStream oos = new ObjectOutputStream(
```

```
                new BufferedOutputStream(zip));
        oos.writeObject(prices);
        oos.close();
        zip.close();
        zippedPrices = baos.toByteArray();
    }

    protected void unzipPrices()
                throws IOException, ClassNotFoundException {
        ByteArrayInputStream bais = new ByteArrayInputStream(zippedPrices);
        GZIPInputStream zip = new GZIPInputStream(bais);
        ObjectInputStream ois = new ObjectInputStream(
                new BufferedInputStream(zip));
        prices = (SortedMap<Date, StockPrice>) ois.readObject();
        ois.close();
        zip.close();
    }
}
```

makeZippedPrices()[2] 将 prices 序列化成字节数组后保存，然后通常在 writeObject() 中调用 defaultWriteObject() 进行序列化。（事实上，如果可以定制序列化，将 zippedPrices 数组变成 transient 直接序列化数组的长度和字节会好一些。不过这个代码示例要清楚一点，且简单一些也更好。）在反序列化时，操作反过来执行。

如果目标是序列化成字节流（就像原先的示例代码一样），这就是个糟糕的提议。这并不令人惊奇，因为压缩字节所需的时间大大超过了写入本地字节数组的时间。参见表 10-12。

表10-12：10 000个对象序列化和反序列化、带压缩和不带压缩时所用时间的对比

场景	序列化时间（秒）	反序列化时间（秒）	数据大小（字节）
无压缩	12.1	8.0	41 170
压缩 / 解压缩	26.8	12.7	5849
仅压缩	26.8	0.494	5849

表中最有趣的是最后一行。在该轮测试中，数据在发送前进行了压缩，但 readObject() 并没有调用 unzipPrices()，而是依据需要，在客户端首次调用 getPrice() 时才调用该方法。readObject() 不再调用 unzipPrices() 后，就只有几个 BigDecimal 对象需要反序列化，速度非常快。

在这个例子中，很可能会出现客户端永远不需要实际的股票价格的情况：客户端可能只需要调用 getHighPrice() 和类似的方法获取合计数据。如果所有方法都是只在需要时获取数据，那么延迟解压价格数据信息就能节省大量时间。如果对象可能需要持久化，延迟解压也会有用（比如，备份 HTTP 会话状态，以防应用服务器失败）。延迟解压既节约 CPU 时间（因为跳过了解压），也节约内存（因为压缩后的数据需要的内存空间更小）。

所以，即便应用在高速局域网络中运行——尤其当目标是节约内存而不是时间时——对序列化数据进行压缩并延迟解压也仍然很有用。

注 2：原文是“zipPrices”，代码中并没有此方法。——译者注

如果序列化是为了在网络中传输，那任何数据压缩都会有益处。表 10-13 同样是对 10 000 个股票对象进行序列化，不过这次它将数据传向了另一个进程。这个进程可以是在同一个机器上，也可以在通过宽带连接访问的其他机器上。

表10-13：10 000个对象的网络传输时间对比

对象	同一机器（秒）	宽带广域网（秒）
无压缩	30.1	150.1
压缩 / 解压缩	41.3	54.3
只压缩	28.0	44.1

同一机器上的两个进程之间的网络通信是最快的——虽然通信数据会发送到操作系统层，但压根不用通过网络。即便在这种情况下，压缩数据和延迟解压的性能仍然是最快的（至少在这个测试中是如此——但小数据量还是会有所衰退）。可以预料的是，一旦网络速度比较慢，传输数据又有数量级上的差别，总的耗费时间就会有巨大的差别。

10.5.4　追踪对象复制

本节先介绍一个示例，如何不对对象引用进行序列化，以避免在反序列化时处理对象引用。然而，`writeObject()` 中最有力的优化是不重复输出对象引用。在 `StockPriceHistoryImpl` 中，这意味着不重复输出 `prices` map。因为示例采用标准 JDK 中的 map，JDK 的类已经对数据的序列化进行了优化，所以我们不用担心。不过，了解这些类如何进行优化、理解哪些可能的优化都是有益处的。

`StockPriceHistoryImpl` 中的关键结构是 `TreeMap`。图 10-2 是一个简化版本的 map。JVM 默认先序列化 Node A 的原生数据字段，然后递归调用 Node B 的 `writeObject()`（接着是 Node C）。Node B 也会序列化它自己的原生数据字段，然后递归序列化它上级 Node 的字段。

但是请注意——Node B 上级节点 Node A 已经被序列化，怎么办？对象序列化的代码很智能：它会意识到这一点，并且不会再次序列化 Node A 的数据。相反，它只会在先前序列化的数据中添加一个对象引用。

追踪上一级对象从而递归所有对象，会对序列化的性能有少许影响。但正如 `Point` 数组的例子所示，这是无法避免的：必须追踪上一级序列化的对象以便正确恢复对象引用。不过，可以通过压缩对象引用来进行智能优化，从而在对象反序列化时易于重建。

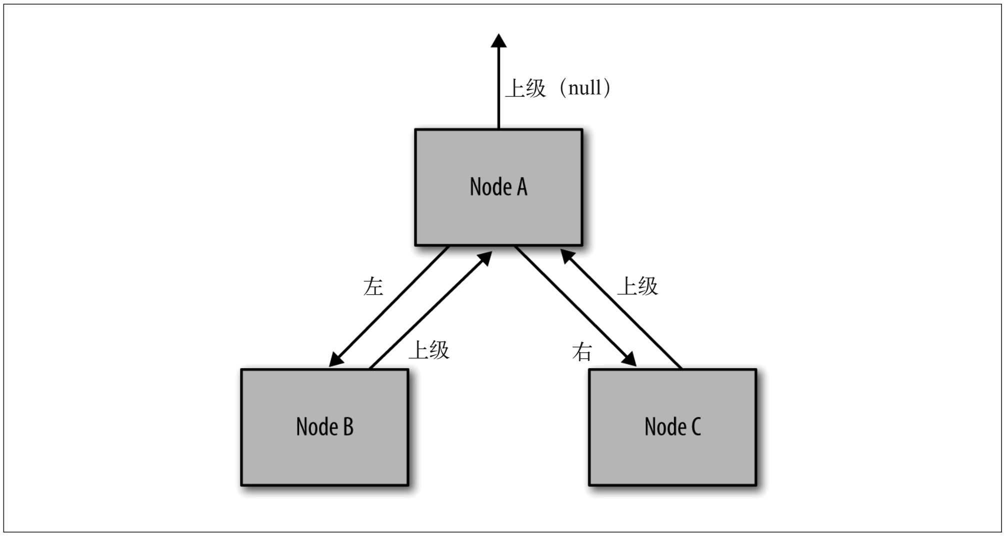


图 10-2：简单的 TreeMap 结构图

不同的集类处理这个问题的方式有所不同。比如 `TreeMap`，它只是遍历树然后序列化键值，丢弃了键之间的所有关系（也就是它们的排列顺序）。在反序列化时，`readObject()` 会重新排列数据并生成树。虽然排序对象听起来代价很昂贵，但实际并非如此：对 10 000 只股票而言，整个过程要比默认的序列化快 20%，默认机制需要追踪所有的对象引用。

需要序列化的对象减少了，因此 `TreeMap` 也能从优化中获益。map 中的 Node（在 JDK 中为 `Entry`）包含两个对象：键和值。由于 map 不会包含两个相同的 Node，所以序列化保留 Node 的对象引用时不用担心。在这种情况下，它不会序列化 Node 对象本身，而是直接序列化键和值。所以最终的 `writeObject()` 看起来像这样（为便于阅读，代码作了调整）：

```
private void writeObject(ObjectOutputStream oos) throws IOException {
    ....
    for (Map.Entry<K,V> e : entrySet()) {
        oos.writeObject(e.getKey());
        oos.writeObject(e.getValue());
    }
    ....
}
```

这段代码看起来与 `Point` 示例中的那段不能正常工作的代码非常像。差别在于该段代码会序列化相同的对象。`TreeMap` 不会有两个相同的 Node，所以没有必要序列化 Node 引用。`TreeMap` 可以有相同的值，所以值必须序列化成对象引用。

这就回到了起点：正如我在本节开头所说的，正确优化对象序列化非常困难。但当对象序列化成为应用的主要瓶颈时，恰当地进行优化可以带来很大的益处。

关于 Externalizable 接口

本节没有讨论另一种优化对象序列化的方法，即实现 Externalizable 接口而不是 Serializable 接口。

实际上，这两个接口的差别在于它们如何处理非 transient 字段。当 writeObject() 调用 defaultWriteObject() 时，Serializable 会序列化非 transient 字段。但 Externalizable 没有这样的方法。Externalizable 类必须显式序列化所有关注的字段，无论 transient 与否。

即便一个对象中的所有字段都是 transient，也最好实现 Serializable 接口，并调用 defaultWriteObject() 方法。这使得代码在添加（移除）字段时更容易维护。从性能的角度来看，Externalizable 并没有特别的优点：最终影响性能的是数据量的大小。

快速小结

1. 数据的序列化，特别是 Java EE 中的序列化，有可能是很大的性能瓶颈。
2. 将变量标记为 `transient` 可以加快序列化，并减少传输的数据量。这些做法都可以极大地提高性能，除非接收方重建数据需要花费很长时间。
3. 其他 `writeObject()` 和 `readObject()` 方法的优化也可以显著加快序列化。但请小心，因为这容易出错，而且不留神就会引入不易察觉的 bug。
4. 通常在序列化时进行压缩都有益处，即便数据不在慢速网络上传输。

10.6 Java EE网络API

我们前面介绍过的几种数据交换技术——XML 解析、JSON 处理和对象序列化——可以在不同的应用中使用，但主要是在 JAX-WS、JAX- RS 和 IIOP/RMI 这三个 Java EE 网络 API 中使用。

这些 API 的协议差别很大，特性也有很大的不同。你在决定为何使用它们以及何时使用它们时，考虑的首要因素就是它们的特性。关于它们有许多争论，比如 JAX-RS 是否比 JAX-WS 快，不过这些争论都假设有这样一个通用型应用，这个应用可以用两种框架编写。如果确定需要安全特性，那就应该选择 JAX-WS，而不管它与 JAX-RS 相比性能如何。如果应用必须和已有的输出 IIOP 接口的服务器通信，那么选择也就显而易见了。

但对于这几种网络 API 来说，它们需要克服的性能问题类似。本节讨论其中的一些困难以及如何处理这些困难。

调整传输数据的大小

影响这些技术性能的首要因素是数据交换，这是为什么本章花那么多时间讨论它的原因之一。传输的数据量应该尽量小，无论是压缩或去冗，或者其他技术。

另一方面，发起一次网络调用会有很显著的网络开销。设计网络 API 时，应该设计成“粗

粒度”的——也就是说，最好一次调用返回大量数据，使得客户端发起的网络调用总数最小。这个原则与减少数据交换量相违背，所以必须做些权衡。

可以在前面股票 RESTful Web 服务平均响应时间的测试中观察到这种平衡。可以将服务设计成只返回一段时期内的基本数据（最高价、最低价、平均价和标准差），也可以设计成在返回这些基本数据的同时再加上这期间每天的日数据。

如果预先知道客户端要如何使用数据，就能很容易且精确地知道会返回哪些数据。但是，实际情况并不总是这样。在这个例子中，比如说客户端请求某只股票 5 年的历史数据，那开始时客户端应用只会向用户展示概要数据。如果用户想深入了解数据，查看单个股票的日数据，那会发生什么？是否所有数据都应该在第一次调用中返回给客户端，使得查看详情时无需再次发起网络调用？是否应该只返回概要数据，而在用户想要了解某年详情时，程序才再次发起调用获得那年的日数据？是否第二次调用应该获取整个 5 年的历史数据，即便此时用户只想看第三年的数据？

为了确定此处该用哪种策略，可以比较一下返回全部数据的时间和多次网络调用的时间。表 10-14 显示了在不同场景下获取数据的平均响应时间。

(1) 客户端请求 1 年的数据。
(2) 客户端请求 1 年的概要数据。
(3) 客户端请求 5 年的数据。
(4) 客户端发起 2 次请求：先请求概要数据，然后进一步请求特定时间的详细数据。
(5) 客户端发起 10 个请求：1 个概要请求，加上 9 个特定时间详细数据的请求。

这些测试是在我的宽带连接下进行的。毫无疑问，网络速度对报告的时间有巨大影响（参见表 10-14）。

表10-14：各种场景下RESTful的平均响应时间

场景	平均响应时间（毫秒）	数据量（字节）
1 年数据	90	30 K
1 年概要数据	30	60
5 年数据	300	186 K
2 次概要请求	60	2 次调用；每次 60
10 次概要请求	280	10 次调用；每次 60

获取一整年完整数据的时间并不比只获取概要数据长太多，所以如果用户只需要获取其中的三个数据片段，那一次性返回整个数据集总是更好的选择。

不过 5 年的概要数据有点不同：数据编组和发送所需的时间要长得多，所以在总时间达到平等前，用户需要发起 11 个获取详情的请求。

本例中的时间包括将 RESTful 服务返回的 JSON 数据编组的时间，这个时间取决于数据有多少年。但是多个客户端可能会请求相同的数据集，这种情况可以重用之前编好组的数据。如果编好组的数据已经计算好，响应时间的拐点就会有很大的不同，如表 10-15 所示。

表10-15：各种场景下（响应有缓存）RESTful的平均响应时间

场景	平均响应时间（毫秒）
1 年数据	50
1 年概要数据	30
5 年数据	90
2 次概要请求	60
10 次概要请求	270

由于调用一次概要数据的开销基本不变，所以概要数据的响应时间差别很小。而获取 1 年和 5 年完整数据的时间只是在传输数据上，明显少于之前数据需要计算和编组的情况。一般来说，可以返回给客户端大量可能并不需要的数据，而没有太多性能上的损耗。

网络已今非昔比

虽然我看起来不太像老顽固，但我确实曾经用 300 波特的调制解调器连接过远程网络。（幸运的是，我不必穿过 4 英尺厚的雪步行 5 英里到学校。）对于这种情况，传统上的偏见会认为有许多的网络调用。

如今，我打开浏览器，每次敲入一些字符时，都会调用远程的 Google 服务器，然后它会返回给我一些文字，辅助我进行一些搜索。Google 已经算出，完成调用大约需要 200 毫秒或更少，这种延迟时间并不会引起我的注意。

随着网络速度越来越快，本节我所讨论的权宜之计会变得越来越不重要。另一方面，我的智能手机大多数时候都有幸能使用快速的 3G 信号，一年中少数时候我在海洋上，我也很有幸可以通过卫星连接互联网。更快的硬件（或网络）总是有助于性能的提升，但却不能以此替代性能设计。如果应用可以检测出正处于快速网络，就能进行快速细致的网络调用，这会是一个很好的特性。要确保应用可以正常工作，无论它部署在哪里。

10.7　小结

Java EE 应用的性能依赖于好几个因素。其中应用代码的质量永远是最主要的。而且，因为用到了许多外部资源，所以应用服务器的性能瓶颈通常并不在 Java 层。

许多影响应用服务器性能的因素并不只是针对 Java EE——尤其是线程的性能、对象池和网络性能。对应用服务器而言，最重要的影响因素是它传输或处理的数据量——无论是简单的 HTML，还是 XML、序列化对象状态或 JSON 等。本章列出的实践有助于你最大限度地发挥可用资源对应用服务器的作用。

第 11 章

数据库性能的最佳实践

本章主要探讨由 Java 驱动的数据库应用的性能。访问数据库的应用程序经常会受制于一些与 Java 不直接相关的性能问题，譬如使用的数据库有 I/O 瓶颈，或者由于关键索引缺失，导致 SQL 查询需要做全表扫描。要解决这些问题，不必调优 JVM，也不需要修改应用代码。处理数据库相关的性能问题需要我们（从其他的源头）了解如何为数据库应用编程及调优。

但这并不是说，使用数据库的应用的性能与 JVM 管理的对象及使用的 Java 技术没有多少相关性。恰恰相反，为了获得更优的性能，我们要确保数据库和应用都得到最好的调优，运行于最佳的代码路径之上。

> **示例数据库**
>
> 本章的例子使用了一个样本数据库，存储了 128 支股票全年的交易数据。一年中有 261 个工作日。
>
> 每支股票的交易价格保存在一张名为“STOCKPRICE”的表内，这张表的主键分别是股票代号和交易日期。所以这张表中有 33 408 条记录（即 128 × 261）。
>
> 每一支股票都有由五个相关的期权构成的集合，这些期权也都是每日订价。“STOCKOPTIONPRICE”表保存的数据包括股票代码、日期和代表期权编号的整型数，其中股票代码是主键。所以这个表包含 167 040 行数据（即 128 × 261 × 5）。

11.1 JDBC

本章从 Java 持久化 API（Java Persistence API，JPA）的角度讨论数据库的性能——JPA 当

前的版本是 2.0。然而，JPA 的底层调用也是 JDBC，很多程序员在编写程序时还是喜欢在应用代码中直接调用 JDBC 接口——因此，这也是我们调查 JDBC 性能问题时重要的一方面。即使应用程序已经使用了 JPA（或者其他的数据库框架），理解哪些方面会影响 JDBC 的性能对我们最大限度地提升框架性能的目标也是相当有益的。

11.1.1 JDBC驱动程序

JDBC 驱动程序是影响数据库应用程序性能的诸多因素中最重要的一个。几乎每种数据库都提供了自己的 JDBC 驱动程序，大多数流行的数据库往往还有第三方厂商的 JDBC 驱动程序可用。通常情况下，使用第三方驱动程序的原因都是它们能提供更好的性能。

我无法对所有数据库驱动程序的性能表现作裁决，但是我们可以在这里探讨评估不同数据库驱动程序时应该考虑的因素。

1. 工作于何处进行

JDBC 驱动程序的实现有两种选择，要么在 Java 应用端（即数据库的客户端）完成更多的工作；要么将工作推迟到数据库服务器端进行。阐释这个问题最合适的例子是 Oracle 数据库的“瘦驱动程序”和“胖驱动程序”。瘦驱动程序的实现对 Java 应用程序自身的影响极小，它依赖数据库服务器进行更多的处理工作。胖驱动程序的实现与此相反：它将数据库的工作迁移到 Java 客户端来完成，代价是客户端会消耗更多的处理能力和更多的内存。大多数的数据库都提供了类似的方案。

关于哪种模式能提供更好的性能，有诸多的争论。实际上，这两种模式都不能提供以不变应万变的优势——即到底采用哪种驱动程序能获得最好的性能往往取决于具体的运行环境。譬如，应用程序运行的服务器比较弱，是个双核的机器，它连接的是一个性能强大、调优良好的数据库服务器，那么在这种情况下，很可能应用还没施加多大的压力到数据库，应用服务器自身的 CPU 就已经达到了 100%。这种情况下，使用“瘦客户端”类型的驱动程序往往能获得更好的性能。与此相反，在一个大型企业中，100 多个部门都需要访问人力资源数据库时，尽量保留数据库服务器的资源，部署“胖客户端”型的驱动程序才能获得更好的性能。

这就是我们谈论 JDBC 驱动程序的性能时要保持谨慎怀疑态度的原因：经常会出现这样的情况，即某个驱动程序在特定的环境下表现的性能优于其他供应商，但在你自己的环境中，采用同样的配置效果却不尽人意。尽量在你自己的环境中进行测试，务必确保你的测试环境与你将来部署的生产环境保持一致。

2. JDBC驱动程序的类型

JDBC 驱动程序可以划分为四种类型（1–4）。目前使用最广泛的是 2 型（使用本地代码）和 4 型（纯 Java 代码）。

1 型驱动程序在 ODBC 和 JDBC 之间搭建了一座通信的桥梁。如果应用程序需要访问 ODBC 的数据库，只能使用这种类型的驱动程序。1 型驱动程序通常性能都不好；尽量避免使用 ODBC。

3 型和 4 型的驱动程序都采用纯 Java 语言编写，但是 3 型的驱动程序通常是为特定的架构

而设计的，这些架构中，中间件的某些部分（有时也可能是应用服务器，不过绝大多时候都不是）会对中间结果进行翻译。在这种架构中，JDBC 客户端程序（通常是独立的程序，不过最常见的还是应用服务器）向中间件发送 JDBC 协议报文，中间件解析这些请求，将其转译为数据库协议，并转发到数据库服务器（对数据库返回的数据采用逆向的转译）。

这种架构有时是必不可少的：中间件部署在网络的非军事区（Demilitarized Zone，DMZ），为访问数据库的连接提供额外的安全保护。从性能的角度看，这种部署方式既有其优势也有其弊端。中间件可以对数据库中的记录进行缓存，而这能减轻数据库服务器的压力（在某种程度上让它跑得更快），能够更快地返回客户端请求的数据（降低了请求的延迟）。不过，如果关闭了中间件的缓存，数据库访问的性能就会受到影响，因为每次数据库操作都要经历两次网络往返。理想的情况下，这些因素可以忽略不计（或者由于缓存，数据的请求响应得更快）。

然而在实际生产环境中，这种架构并没有广泛地部署（一如往常，情况总在不断地变化。譬如，甲骨文公司为它的分布式远程连接池（Distributed Remote Connection Pool，DRCP）提供了 JDBC 驱动程序，这是一种 3 型驱动程序，不过它和 4 型 JDBC 驱动程序一样也采用了 JAR 文件作为发布的格式，但是，对于终端用户而言，JDBC 驱动程序到底是 3 型还是 4 型并不是那么重要）。通常情况下，将应用服务器部署到中间层（如果有必要，就部署到 DMZ 之内）是更容易的方案。这样，应用服务器既能完成正常的数据库操作，又不必为客户端提供任何 JDBC 的接口：由于提供了诸如 servlet 接口、Web Service 接口等等这样的服务，架构的灵活性增强了，客户端不再需要了解任何数据库内部的细节。

这之后剩下的是 2 型和 4 型驱动程序，但是跟其他的驱动程序比起来，这二者中的任何一个都不具备与生俱来的性能优势。2 型驱动程序会受 JNI 引入的开销影响，但是设计良好的 2 型驱动程序能避免这样的问题。不要将驱动程序的类型（2 型或 4 型）和前文介绍过的驱动程序的特征——它是“胖驱动程序”还是“瘦驱动程序”混为一谈。虽然 2 型驱动程序往往具有胖驱动程序的特征，但 4 型驱动程序通常表现出瘦驱动程序的特点，但是这并非严格意义上的要求。最后，选择 2 型驱动程序还是 4 型驱动程序，取决于你的环境以及具体的驱动程序。并没有一种先验的方法能判断哪种方式一定能获得更好的性能。

快速小结

1. 花时间评估挑选出最适合你的应用程序的 JDBC 驱动程序。
2. 最合适的驱动程序往往依特定的部署环境而有所不同。对同样的应用程序，在一个部署环境中可能要使用 JDBC 驱动程序，在另一个部署环境中则要采用不同的 JDBC 驱动程序，才能有更好的性能。
3. 如果可以选择，尽量避免使用 ODBC 和 JDBC 1 型的驱动程序。

11.1.2 预处理语句和语句池

大多数情况下，代码中若要进行 JDBC 调用，推荐使用 `PreparedStatement`，尽量避免直接使用 `Statement`。这二者的区别在于预处理语句让数据库有机会重用已经执行过的 SQL 信息。而这能够帮助节省之后运行的预处理语句的开销，提升执行效率。

“重用”这个词用在这儿是相当贴切的：首次使用预处理语句时数据库会耗费更多的执行时间，因为它需要设置和保存相应的信息。如果这个语句仅使用一次，这些额外的工作就被浪费了，这种情况下，使用常规的语句（`Statement`）可能是更好的选择。

对只进行少量数据库调用的批处理类型（batch-oriented）的程序，选择 `Statement` 接口能让程序更迅速地完成工作。但是批处理类型的程序也可能发起成百上千的 JDBC 调用，不过调用的对象都是几乎一样的几个 SQL 语句；后者就是本章要讨论的对象，它使用批处理程序加载一个含有 10 000 条股票记录的数据库。需要处理大量 JDBC 调用的批处理程序，包括生命周期内要服务大量请求的应用服务器，使用 `PreparedStatement` 接口都能取得较好的效果（JPA 这样的数据库框架会自动采用这样的接口）。

对预处理语句进行池化（pooled），才能体现出预处理语句的性能优势——即实际生产中，重用 `PreparedStatement` 对象才能更显著地提升性能。为了更好地进行“池化”，我们需要注意两件事，即 JDBC 连接池以及 JDBC 驱动程序的配置（语句池根据数据库供应商的不同，也常常被称作语句缓存，`Statement Caching`）。这些配置选项适用于任何使用 JDBC 的应用程序，无论它们是直接使用 JDBC 驱动程序，还是以 JPA 的方式间接调用 JDBC。

1. 设置语句池（statement pool）

预处理语句池基于每个连接进行工作。如果程序的一个线程从池内取得一个 JDBC 连接，并以此为基础创建一个预处理语句，那么所有跟这个语句相关的内容都仅在该连接的生命周期内有效。第二个线程会使用新的连接创建自己的预处理语句连接池实例。最终，每个连接对象都会创建自己的预处理语句池，服务于应用程序（假设它们在应用程序的生命周期中会一直被使用）。

这就是为什么独立的 JDBC 应用应该使用连接池的原因之一（JPA 会为 Java SE 的应用程序创建一个连接池，并且这个过程是透明的，Java EE 环境中，程序可以使用应用服务器的连接池）。这也意味着连接池的大小是非常重要的（无论是使用 JDBC 还是使用 JPA 的程序）。在程序的早期运行中，这一点尤其重要：如果连接没有使用预处理语句，第一个请求的处理就会比较慢。选择合适的连接池大小也非常重要，因为连接池要缓存预处理语句，而这些预处理语句的缓存会占用堆空间（并且通常是大量的堆空间）。这种情况下，对象重用当然是件好事，但是你还需要考察重用的对象要占用多少堆空间，以避免对 GC 时间产生负面的影响。

2. 语句池的管理

使用预处理语句池时第二件要特别注意的事是哪些代码会实际地创建和管理池。预处理语句池在 JDBC 3.0 中首次引入，它提供了一个方法（即 `ConnectionPoolDataSource` 类的 `setMaxStatements()` 方法）用于开启和禁用语句池。如果传递给 `setMaxStatements()` 方法的参数是 0，语句池就被禁用。这个接口并未明确定义语句池在什么地方创建——是在 JDBC 驱动程序层面还是在其他的层面上，譬如应用服务器上。这个单一的接口无法适配所有的 JDBC 驱动程序，有些情况还需要额外的配置才能满足需要。

因此，编写直接进行 JDBC 调用的 Java SE 应用时，你有两个选择：要么通过配置 JDBC 驱动程序来创建和管理语句池，要么在应用程序代码中创建和管理语句池。Java EE 的应用有两种可能性（略微不同）：由 JDBC 驱动程序来创建和管理语句池，或者由应用服务器来创建和管理语句池。

这里最麻烦的问题是这个领域没有统一的标准。有的 JDBC 驱动程序根本没有提供语句池的机制，它们的开发者希望能简化驱动程序的结构，由应用服务器来完成语句池的管理。有的应用服务器并未提供语句池管理的功能；它们希望由 JDBC 驱动程序来进行这方面的工作，不愿意覆盖复杂的语句池管理。这两种观点都有一定的道理（不过，如果 JDBC 驱动程序不提供语句池管理机制，这副重担就落在了应用程序开发者的肩上）。最终，你要厘清不同方案的利弊，确保语句池在某个地方创建。

由于没有公认的标准，你可能会碰到 JDBC 驱动程序和应用服务器同时支持预处理语句池的情况。这时，确保只有一方在管理预处理语句池是非常重要的。从性能的角度出发，更好的选择取决于驱动程序和应用服务器的组合。通用的原则是，我们可以期望 JDBC 驱动程序在语句池的管理上性能更优。这是因为驱动程序（通常情况下）是针对特定的数据库开发的，会针对这种数据库进行相应的优化，而更通用的应用服务器代码很难做到这一点。

想要了解如何开启某个 JDBC 驱动程序的语句池（或者缓存池）功能，可以查询驱动程序的文档。很多时候，你只需要将驱动程序的 `maxStatements` 属性设定为期望的值（即语句连接池的大小）就完成了配置的工作。另一些驱动程序可能还需要进行额外的设置（譬如，Oracle 的 JDBC 驱动程序就需要设置某些属性，并依此决定使用隐式的语句池还是显示的语句池）。

快速小结

1. Java 应用程序通常都会重复地运行同样的 SQL 语句。这些情况下，重用预处理语句池能极大地提升程序的性能。
2. 预处理语句必须依单个连接进行池化。大多数的 JDBC 驱动程序和 Java EE 框架默认都提供了这一功能。
3. 预处理语句会消耗大量的堆空间。我们需要仔细调优语句池的大小，避免由于对大量大型对象池化而引起 GC 方面的问题。

11.1.3 JDBC连接池

创建数据库连接是非常耗时的操作，因此，JDBC 连接是 Java 程序中应该尽量重用的典型对象。

在 Java EE 的环境中，所有的 JDBC 连接都源自应用服务器的连接池。在使用 JPA 的 Java SE 环境中，大多数的 JPA 提供商都默认使用一个连接池，你可以通过配置文件 `persistence.xml` 对连接池进行自定义。单机版的 Java SE 环境里，连接是由应用程序来控制的。对于最后一种情况，你可以从连接池的第三方库中选择其一进行构建，这些库从很多渠道都能找到。不过，对于单独的应用程序，很多时候，最简单的方式是为每个线程创建一个连接，将它保存在线程的本地变量中。

一如往常，平衡池化对象的内存占用以及由于池化触发的额外 GC 数量是非常重要的。由于预处理语句缓存的开销，这一点变得尤其要紧。实际的连接对象可能不是非常大，但是语句缓存（它们建立在每个连接的基础之上）可以变得非常大。

这种情况下，数据库也需要寻求适当的平衡。每个数据库的连接都需要数据库分配资源（此外，应用程序还要消耗一定的内存）。随着连接不断地添加到数据库，数据库需要的资源越来越多：它会为每个 JDBC 驱动程序使用的预处理语句分配更多的内存。如果应用服务器开启了过多的连接，数据库的性能会受到负面的影响。

对于连接池而言，首要的原则是应用的每个线程都持有一个连接。对应用服务器而言，则是初始时将线程池和连接池的大小设置为同一值。对单一的应用程序，则是依据应用程序创建的线程数调整连接池的大小。典型的情况下，这种设置能取得最好的性能：所有的程序线程都不需要等待数据库连接的释放，数据库亦有足够的资源处理来自应用程序的负荷。

然而，如果数据库成为瓶颈，采用这条规则可能会适得其反。向一个太弱的数据库施加过多的连接是对前一原则的另一种阐释，即向一个已经非常繁忙的系统增大负荷只会降低它的性能。这种情况下，使用连接池限制施加到数据库的负荷可以改善程序的性能。这时应用程序线程需要等待空闲连接，不过如果数据库没有被压垮的话，总的系统吞吐量会提升。

快速小结

1. 数据库连接对象初始化的代价是昂贵的。所以在 Java 语言中，它们通常都会采用池技术进行管理——要么是通过 JDBC 驱动程序自身管理，要么在 Java EE 和 JPA 框架中进行管理。
2. 跟其他的对象池一样，对连接池的调优也是非常重要的，我们需要确保连接池不会对垃圾收集造成负面的影响。为了达到这个目标，调优连接池，避免对数据库自身的性能产生负面影响也是非常有必要的。

11.1.4 事务

应用程序拥有合理的需求，这些需求最终可以指导事务如何被正确地处理。要求“重复读”（repeatable-read）语义的事务会比只要求“提交读”（read-committed）语义的事务慢，但是了解这些对于无法接受“非重复读”的应用程序而言没有太多的实际参考意义。因此，虽然这一节讨论的主要是如何替应用程序构造使用最小隔离集的语义，但也不要一叶障目地试图用高性能掩盖程序的确切需求。

数据库的事务有两类性能代价。首先，数据库事务的设置和提交都会耗费时间。这个过程包括确保对数据库的修改都已经完全保存到磁盘，检查数据库事务日志的一致性，等等。其次，数据库事务进行期间，通常事务都要对部分数据加锁（不一定总是行锁，不过这里我们会以行锁作为例子）。如果两个事务在同一个数据库行锁上发生竞争，应用的可扩展性就会受影响。从 Java 语言的角度看，这种行为跟第 9 章讨论的竞争锁及非竞争锁很相似。

为了优化性能，我们需要考虑这些问题：如何调整程序，优化事务的实现，让事务自身更加高效，以及在事务中如何持有锁，才能让应用程序的整体性能有更好的扩展性。

1. JDBC事务的控制

本节对事务的介绍包括两种形式，同时涵盖使用 JDBC 和 JPA 的应用，不过 JPA 管理事务的方式不大一样（本章后续内容会详细探讨这些细节）。对于使用 JDBC 应用，事务的开

始和结束都基于如何使用 Connection 对象。

在基础的 JDBC 使用中，连接提供了自动提交（autocommit）模式（通过 setAutoCommit() 方法设置）。如果开启自动提交模式（对大多数的 JDBC 驱动程序，这个选项是默认开启的），则 JDBC 程序中的每个语句自身都是一个事务。这种情况下，程序不需要额外提交事务（事实上，如果调用了 commit() 方法，性能反而可能会恶化）。

如果自动提交被关闭，那么事务默认于连接对象第一次调用时开始（譬如，调用 executeQuery() 方法时）。事务持续运行直到 commit() 方法（或者 rollback() 方法）被调用。新的事务在下一次数据库调用连接操作时开始。

事务的提交操作是非常昂贵的，因此我们的目标之一是尽可能在一次事务中完成更多的工作。不幸的是，这个目标与我们的另一个目标几乎是南辕北辙：因为事务需要持有锁，所以它们应该在尽可能短的时间内完成。很明显，我们需要在二者之间进行权衡，如何进行平衡取决于应用程序以及它的锁需求。下一节探讨事务隔离及锁时会对此作进一步讨论，这里首先看看事务处理自身有哪些优化选项。

下面这段代码用于向数据库中插入数据供股票应用使用。为使每天的数据有效，一行数据会被插入到 STOCKPRICE 表，五行数据插入到 STOCKOPTIONPRICE 表。为了完成这个目标，一个基本的循环如下所示：

```
Connection c = DriverManager.getConnection(URL, user, pw);
PreparedStatement ps = c.prepareStatement(insertStockSQL);
PreparedStatement ps2 = c.prepareStatement(insertOptionSQL)) {
Date curDate = new Date(startDate.getTime());
while (!curDate.after(endDate)) {
    StockPrice sp = createRandomStock(symbol, curDate);
    if (sp != null) {
        ps.clearParameters();
        ps.setBigDecimal(1, sp.getClosingPrice());
        // 其余字段使用类似的集调用
        ps.executeUpdate();
        for (int j = 0; j < 5; j++) {
            ps2.clearParameters();
            ps2.setBigDecimal(1, ...);
            // 其余字段使用类似的集调用
            ps2.executeUpdate();
        }
    } //其余的curDate是周末,故被略过
    curDate.setTime(curDate.getTime() + msPerDay);
}
```

如果开始和结束的日期表示的是 2013 年，那么这个循环会插入 261 行记录到 STOCKPRICE 表（通过第一次调用的 executeUpdate() 方法），并插入 1305 行记录到 STOCKOPTIONPRICE 表（通过内部的 for 循环）。若使用默认的自动提交模式，就意味着会有 1566 个相互独立的事务，而这将是相当昂贵的。

如果关闭自动提交模式，在循环结束时显式地调用提交操作能够获得更好的性能：

```
Connection c = DriverManager.getConnection(URL, user, pw);
c.setAutoCommit(false);
```

```
...
while (!curDate.after(endDate)) {
    ...
}
c.commit();
```

从逻辑的观点来看，这可能也更合理：数据库中要么保存一整年的数据，要么就没有任何数据。

如果这个循环被用于多个股票的计算，就会又面临一个选择，即是一次性提交所有的数据，还是一次提交一年的数据：

```
Connection c = DriverManager.getConnection(URL, user, pw);
c.setAutoCommit(false);
for (int i = 0; i < numStocks; i++) {
    curDate = startDate;
    while (!curdate.after(endDate)) {
        ...
    }
    //c.commit(); // 一次提交一年的数据
}
c.commit(); // 一次性提交所有的数据
```

一次性提交所有的数据能获得最高的性能，这就是注释排除其他选项的原因。不过，这个例子中，还有一种合理的情况，在其场景下应用的语义可能要求单独提交每一年的数据。有时候，其他的需求可能会与获取最佳性能的尝试发生冲突。

每次 executeUpdate() 方法在上述代码中运行时，数据库就会进行一次远程方法调用，执行一些工作。此外，更新表时，数据会被加锁（至少要确保另一个事务不能使用同样的符号和日期向表内插入记录）。这个例子中，通过批量插入可以进一步优化事务的处理。插入批量化时，JDBC 驱动程序会保持这些数据，直到批量插入完成；之后所有的语句会被发送到一个远程 JDBC 调用中。

下面展示了批量处理是如何实现的：

```
for (int i = 0; i < numStocks; i++) {
    while (!curdate.after(endDate)) {
        ...
        ps.addBatch();  // 替换executeUpdate()调用
        for (int j = 0; j < 5; j++) {
            ...
            ps2.addBatch();  // 替换executeUpdate()调用
        }
    }
}
ps.executeBatch();
ps2.executeBatch();
c.commit();
```

这段代码同样还能用于依据每支股票进行批量执行（在 while 循环之后）。有些 JDBC 驱动程序对它们能够批量执行的语句数做了限制（批量处理的确会消耗应用程序的内存），所以，即使可以在整个操作完成之后再进行提交，批量处理还是需要频繁地执行。

这些优化措施对性能的提升是非常明显的。表 11-1 展示了将 128 支股票一年的数据插入数据库所花费的时间（总计 200 448 次插入操作）。

表11-1：插入128支股票数据的耗时

编程模式	消耗时间（秒）	DB访问次数	DB提交次数
启用自动提交，不使用批处理	2220.53	200 448	200 448
每支股票提交一次	174.44	200 448	128
所有数据一次性提交	169.34	200 448	1
每支股票的每次提交都用一次批处理完成	19.32	128	128
每支股票都用一次批处理完成，1 次提交	17.33	128	1
所有的股票用一次批处理完成，1 次提交	11.55	1	1

注意，这张表中有一个有趣的事实，不过并不明显：第一行和第二行的差别在于自动提交被关闭了，代码在每个 `while` 循环的结尾显式地调用 `commit()` 方法完成提交。第一行和第四行的差别是语句被批处理（batched）了——不过自动提交还是开启的。一次批处理调用被当成一个事务，这也是为什么数据库调用和提交是一一对应关系的原因。这个例子告诉我们，通过批处理实现的性能提升要明显高于显式地进行事务控制。

2. 事务隔离和锁

影响事务性能的第二个因素与数据库的扩展性相关，因为事务中的数据会被锁定。锁机制能保证数据的完整性，用数据库的术语来说，它让一个事务与其他的事务相互隔离。JDBC 和 JPA 都支持四种主要的数据库事务隔离模式，不过在具体的实现方式上各有不同。

虽然使用正确的隔离模式（Isolation Mode）进行程序设计实际上并不是一个纯 Java 的问题，我们还是会简短地介绍下隔离模式。要了解更多相关信息，请参考数据库编程相关的书籍。

基本的事务隔离模式如下所列（依照开销最大到最小的顺序）。

TRANSACTION_SERIALIZABLE

　　这是最昂贵的事务模式；它要求在事务进行期间，事务涉及的所有数据都被锁定。通过主键访问数据以及通过 WHERE 子句访问数据都属于这种情况：使用 WHERE 子句时，表被锁定，避免事务进行期间有新的满足 WHERE 语句的记录被加入。序列化事务每次查询时看到的数据都是一样的。

TRANSACTION_REPEATABLE_READ

　　这种模式下要求事务进行期间，所有访问的数据都被锁定。不过，其他的事务可以随时在表中插入新的行。这种模式下可能会发生“幻读”（phantom read），即事务再次执行带有 WHERE 子句的查询语句时，第二次可能会得到不同的数据。

TRANSACTION_READ_COMMITTED

　　使用这种模式时，事务运行期间只有正在写入的行会被锁定。这种模式可能会发生“不可重复读”（nonrepeatable read），即在事务进行中，一个时间点读到的数据到另一个时间点再次读取时，就变得完全不同了。

TRANSACTION_READ_UNCOMMITTED

这是代价最低的事务模式。事务运行期间不会施加任何锁，因此一个事务可以同时读取另一个事务写入（但尚未提交）的数据。这就是著名的“脏读”（dirty read）；由于首次的事务可能会回滚（意味着写入操作实际并未发生），因此可能会导致一系列的问题，因为一旦发生这种情况，第二次的事务就是对非法数据进行操作。

数据库都按照自己默认的事务隔离模式进行工作：MySQL 默认使用 TRANSACTION_REPEATABLE_READ；Oracle 和 DB2 默认使用 TRANSACTION_READ_COMMITTED；诸如此类。除此之外，还有很多与具体数据库相关的事务隔离模式变种。DB2 称他们默认的事务模式为 CS（意为游标稳定性，cursor stability），对其他的三种 JDBC 模式亦采用不同的命名。Oracle 数据库不支持 TRANSACTION_READ_UNCOMMITTED 和 TRANSACTION_REPEATABLE_READ 事务类型。

JDBC 语句执行时，使用数据库默认的隔离模式。另一方面，调用 JDBC 连接的 `setTransaction()` 方法也可以控制数据库支持需要的事务隔离级（如果数据库无法支持指定的级别，JDBC 驱动程序会抛出一个异常，或者自动将隔离级升级到它支持的下一个更严格的隔离级）。

TRANSACTION_NONE 和自动提交

JDBC 规范定义了第五种事务模式，就是本节讨论的 TRANSACTION_NONE。理论上，这种事务模式不能通过 `setTransactionIsolation()` 方法设置，因为如果事务已经存在，它的隔离级就不能被设置为 `none`。不过有的 JDBC 驱动程序（最著名的是 DB2）的确允许进行这样的调用（实际上，它甚至允许将默认隔离级设置为 TRANSACTION_NONE）。另一些 JDBC 驱动程序允许在初始化驱动程序时通过属性设置隔离级为 `none`。

严格来说，使用 TRANSACTION_NONE 语义的连接执行语句时是无法向数据库提交数据的：它只能是只读的查询操作。如果有数据写入，必须添加某种锁；否则，如果一个用户使用 TRANSACTION_NONE 语义向一个表写入一个很长的字符串，另一个用户可能仅会看到写入表中的这个字符串的一部分。可能有些数据库会以这种模式运行，不过数量会非常少；最起码，向一张表写入数据的操作应该是原子操作。因此，生产环境中的写操作至少应该使用 TRANSACTION_READ_UNCOMMITTED 语义。

使用 TRANSACTION_NONE 语义的查询是无法提交的，但是如果 JDBC 驱动程序使用 TRANSACTION_NONE 隔离级同时又开启了自动提交，有可能允许写操作。这意味着数据库将每个查询语句当成一个独立的事务。即使如此，由于数据库（很可能）不允许其他的事务看到部分写入的数据，所以实际使用的隔离级是 TRANSACTION_READ_UNCOMMITTED。

对于简单的 JDBC 程序，这就已经足够了。更常见的情况是——尤其是使用 JPA 的时候——程序希望在一个事务内混合使用不同的隔离级。对一个查询我的员工信息并最终据此进行薪资调整的应用程序而言，它需要访问我的员工记录，这部分数据必须被保护：这些数据应该采用 TRANSACTION_REPEATABLE_READ 隔离级。不过同一个事务还可能

需要访问其他表中的数据，譬如保存我的办公室编号的表。对于这部分数据，没有必要在事务中进行锁定，因此访问这些行时当然可以采用 TRANSACTION_READ_COMMITTED 隔离级（甚至是更低的隔离级）。

JPA 让你可以依据每个实体设定锁的级别（当然实体——通常至少——是数据库中的一行）。由于合理设置这些锁的级别是相当困难的，因此相对于直接使用 JDBC 语句设置锁，利用 JPA 让这部分工作变得容易多了。此外，JPA 中采用悲观锁或者乐观锁语义，可以达到与 JDBC 应用中使用不同锁级别同样的效果（如果你对这些语义还不是很熟悉，可通过下面这个例子来熟悉这些概念）。

在 JDBC 层次上，基本的方式是将连接的隔离级设置为 TRANSACTION_READ_UNCOMMITTED，之后按照需要显式地对事务中的数据上锁：

```
Connection c = DriverManager.getConnection(); // 或……从本地池中获取
c.setAutoCommit(false);
c.setTransactionIsolation(TRANSACTION_READ_UNCOMMITTED);
PreparedStatement ps1 = c.prepareStatement(
    "SELECT * FROM employee WHERE e_id = ? FOR UPDATE");
……处理来自ps1的信息……
PreparedStatement ps2 = c.prepareStatement(
   "SELECT * FROM office WHERE office_id = ?");
……处理来自ps2的信息……
c.commit();
```

`ps1` 语句显式地在员工数据表上设置了一把锁：在这次事务进行过程中，其他事务都无法访问这行数据。实现这个目的的 SQL 格式并不是标准的。你需要根据你的数据库提供商的文档，查找获得所需的锁级别的方法，但通常的格式是包含 FOR UPDATE 子句。这种类型的锁被称为*悲观锁*（pessimistic locking）。它有效地阻止了其他的事务，使其不得访问保护的数据。

锁的性能通常可以通过使用*乐观锁*（optimistic locking）的方式改善——这与 `java.util.concurrent.atomic` 解决无竞争原子操作（uncontended atomic operation）的方式异曲同工。如果数据访问不存在竞争，采用乐观锁能显著提升程序的性能。然而，如果数据访问存在竞争，程序设计就变得更加复杂。

对于数据库而言，乐观并发是通过版本列实现的。从一行中选择数据时，选择项必须包含期望的数据及版本列。譬如，为了获取关于我的信息，可以使用下面的 SQL 语句：

```
SELECT first_name, last_name, version FROM employee WHERE e_id = 5058;
```

这个查询会返回我的名字（Scott 和 Oaks）以及当前的版本号（譬如 1012）。事务完成时，会更新当前的版本列：

```
UPDATE employee SET version = 1013 WHERE e_id = 5058 AND version = 1012;
```

如果访问的行要求重复读取或者序列化语义，那么即使事务中只是对数据做读操作也需要更新版本号——这些隔离级要求对事务中的只读数据进行锁定。对于读－提交语义，只有在该行中的其他数据发生更新时才需要更新版本号。

这种模式下，如果两个事务同时使用我的员工记录，则每个都会读取版本号为 1012 的数

据。第一个事务会更新数据的版本号到 1013 并继续运行。此后，第二个事务就无法更新员工记录了——因为表中不再有版本号为 1012 的记录，因此 SQL 更新的语句会失败。该事务会接到一个异常并回滚。

这个例子展示了数据库的乐观锁与 Java 的原子原语之间的主要差异：在数据库编程中，事务收到异常时，是不会进行（也无法进行）透明的重试的。如果你直接对 JDBC 进行编程，`commit()` 方法会收到一个 `SQLException`；在 JPA 中，事务提交时，你的应用程序会收到一个 `OptimisticLockException`。

根据你看问题的角度，这可能是件好事，也可能是件坏事。讨论原子工具集的性能时（它们不会透明地进行重试操作），我观察到冲突频繁的情况下，如果有大量的重试操作，程序的性能会受到极大的影响，因为重试会占用大量的 CPU 资源。对数据库应用来说，这种情况会更加严重，因为事务中的代码运行比简单地增加内存某个区间的值要复杂得多。在数据库应用中，重试失败的乐观事务对性能的影响要大得多，它可能会导致永无止境的重试死循环。除此之外，很多时候我们无法判断哪些操作能自动地进行重试。

因此，不提供自动重试机制也许是件好事（很多时候是唯一可行的方案），但是另一方面，这并不是说应用程序现在不负责处理异常。应用程序可以选择重试事务（可能是一次或者两次），可以选择提示用户请求不同的输入，或者它可以简单地通知用户操作失败。这个问题也没有普遍适用的答案。

如果两个数据源之间极少有机会发生碰撞，则使用乐观锁工作是最好的。假设有一个联合支票账户出现了这样的情况：丈夫和妻子在城市的不同地方，几乎在同一个时刻从这个联合账户中提取现金。这会为我们触发一个乐观锁异常。即使这样的情况发生，要求我们中的一个重试也并不是太繁重的任务，现在发生乐观锁异常的机会就几乎是零了（或者我希望如此：我们不需要解决以多高的频率从 ATM 机上提取现金的问题）。与刚才的场景在某种程度上完全相反的是样本股票的应用。真实世界中，这些数据更新得如此频繁，使用乐观锁只能适得其反。实际上，由于变化的数量庞大，股票应用更频繁使用的是无锁机制，不过实际的交易更新还是会使用某种类型的锁。

快速小结

1. 事务会从两个方面影响应用程序的性能：事务提交是非常昂贵的，与事务相关的锁机制会限制数据库的扩展性。
2. 这两个方面的效果是相互制约的：为了提交事务而等待太长时间会增大事务相关锁的持有时间。尤其是对使用严格语义的事务来说，平衡的方式是使用更多更频繁的提交来取代长时间地持有锁。
3. JDBC 中为了细粒度地控制事务，可以默认使用 TRANSACTION_READ_UNCOMMITTED 隔离级，然后显式地按需锁定数据。

11.1.5　结果集的处理

典型的数据库应用会对一个区间的数据进行操作。譬如股票应用要对某支股票的历史价格进行处理。通过一条 `SELECT` 语句可以将历史记录载入：

```
SELECT * FROM stockprice WHERE symbol = 'TPKS' AND
        pricedate >= '2013-01-01' AND pricedate <= '2013-12-31';
```

这条语句会返回 261 条数据记录。如果还需要对应股票的股价，可以采用类似的查询得到五倍数量的记录。获取样本数据库中所有数据（128 支股票一年的数据）的 SQL 语句会返回 200 448 条数据记录。

```
SELECT * FROM stockprice s, stockoptionprice o WHERE
        o.symbol = s.symbol AND s.pricedate >= '2013-01-01'
        AND s.pricedate <= '2013-12-31';
```

为了使用这些数据，代码需要遍历结果集：

```
PreparedStatement ps = c.prepareStatement(...);
ResultSet rs = ps.executeQuery();
while (rs.next()) {
    ……读取当前的行……
}
```

这里就有一个问题，即这 200 448 条记录保存在什么地方。如果整个结果集的数据都在执行 `executeQuery()` 调用时返回，应用程序就会在它的堆内保存大量的活跃数据，而这可能会导致 GC 或者其他的问题。与此相反，如果只返回调用 `next()` 方法时的一行数据，在处理结果集时，应用和数据库之间就会有大量的往返流量。

与之前一样，这个问题也没有所谓的正确答案；在有的情况下，在数据库中保持大多数的数据，需要时进行提取是更高效的方法；而在另一些场景里，查询时一次性地将所有的数据返回可能会更高效。通过 `PreparedStatement` 对象的 `setFetchSize()` 方法可以控制这些行为，它能通知 JDBC 驱动程序一次返回多少行数据。

这个参数的默认值随 JDBC 驱动程序的不同而异；譬如对 Oracle 的 JDBC 驱动程序，该默认值是 10。在上面展示的循环语句中调用 `executeQuery()` 方法，Oracle 数据库会返回 10 行数据，返回的数据会由 JDBC 驱动程序在内部缓存。头 10 次 `next()` 方法的调用都直接从缓存行中读取，返回其中的一条记录。第 11 次调用该方法时，JDBC 驱动程序会向数据库发出请求，取得另 10 行数据，如此周而复始。

设置提取缓冲区大小的其他方法

前面我们曾推荐使用（预处理）语句对象的 `setFetchSize()` 方法，但是该方法依赖于 `ResultSet` 接口。在任何情况下，设置的提取缓存区的大小都只是建议值。JDBC 驱动程序可以自由决定是要忽略该设置，还是将其圆整为其他的值，亦或是设置为它期望的任何值。没有任何一个方法能一劳永逸地保证设置一定的工作，但是在查询之前设置该参数能获得更多的机会采用你期望的缓冲区大小。

如果连接是通过向 `DriverManager` 的 `getConnection()` 方法传递参数的方式创建，那么有些 JDBC 驱动程序也允许你设置默认的提取缓冲区大小。你可能需要查看你供应商的文档，判断到底哪种方法对你而言更容易。

虽然各 JDBC 驱动程序的默认提取缓冲区大小各有不同，不过它们的典型特征是都相当

小。这种设置在大多数情况下是合理的，尤其是，这种设置不大可能会引起应用程序的内存问题。不过如果 `next()` 方法的性能时不时地非常慢（或者结果集的首次查询方法性能很差），你可能就需要考虑增大提取缓冲区的大小。

快速小结

1. 需要查询处理大量数据的应用程序应该考虑增大数据提取缓冲区的大小。
2. 我们总是面临着一个取舍，即在应用程序中载入大量的数据（直接导致垃圾收集器面临巨大的压力），还是频繁地进行数据库调用，每次获取数据集的一部分。

11.2 JPA

JPA 的性能直接受底层 JDBC 驱动程序的影响，大多数影响 JDBC 驱动程序性能的因素都同样作用于 JPA。除此之外，JPA 的性能还受一些额外因素的影响。

通过调整实体类的字节码能够实现很多的 JPA 性能提升。在 Java EE 的环境中，这种性能的提升是无缝透明的。在 Java SE 的环境中，确保使用正确的字节码的处理方式是非常重要的。否则，JPA 应用程序的性能可能是无法预测的：期望推迟载入的字段可能很早就加载了，保存到数据库的数据可能是冗余的，期望保持在 JPA 缓存中的数据可能还需要从数据库中再次提取，等等。

为 JPA 特别定制的字节码处理方法并不存在。通常情况下，这是作为编译过程的一部分完成的——实体类完成编译后（在它们被载入到 JAR 文件、或者由 JVM 开始运行之前），它们被传递给与具体实现相关的后处理器（postprocessor），对字节码进行“增强”，最终生成一个替换类，这个类按照需要进行了优化。

有的 JPA 实现还提供了对字节码的动态优化机制，可以在类装载到 JVM 的过程中对其进行优化。这种方式需要在 JVM 内部运行一个代理，类载入到 JVM 时，代理会插入到类的载入过程中，在他们被用于定义类之前，对这些字节码进行修改。我们可以在应用程序的命令行指定使用代理，例如对于 EclipseLink，你可以添加 `-javaagent:path_to/eclipselink.jar` 到命令行参数列表中。

11.2.1 事务处理

JPA 同时适用于 Java SE 和 Java EE 的应用。应用运行的平台会影响 JPA 事务的处理方式。

Java EE 中，JPA 事务是应用服务器的 Java 事务 API（Java Transaction API，JTA）实现的组成部分。这种设计提供了两种实现事务的选择：可以由应用服务器来处理边界（使用容器管理事务，即 Container Managed Transactions，CMT），或者由应用程序通过编程显式地控制事务边界（使用用户管理事务，即 User-Managed Transaction，UMT）。

等效使用时，CMT 和 UMT 在性能上没有显著的差异。然而，它们并不总是可以等效地使用的。尤其是跟 CMT 对比起来，UMT 的范畴变化很大，而这些对性能的影响非常显著。我们以下面的伪代码为例讲解说明：

```
@Stateless
public class Calculator {
    @PersistenceContext(unitName="Calc")
    EntityManager em;

    @TransactionAttribute(REQUIRED)
    public void calculate() {
        Parameters p = em.find(...);
        ……进行昂贵的计算……
        em.persist(...answer...);
    }
}
```

这里事务作用的范围（使用 CMT 的情况）是整个方法。如果该方法要求对持久化的数据具备重复读语义，表中的数据在昂贵的计算过程中就会被锁定。

采用 UMT 方式会更加灵活：

```
@Stateless
public class Calculator {
    @PersistenceContext(unitName="Calc")
    EntityManager em;

    public void calculate() {
        UserTransaction ut = ……在应用服务器中查找UT……;
        ut.begin();
        Parameters p = em.find(...);
        ut.commit();
        ……进行昂贵的计算……
        ut.begin();
        em.persist(...answer...);
        ut.commit();
    }
}
```

切分事务可以限制事务对应用程序扩展性的影响，而这只能通过 UMT 方式实现。严格来说，使用 CMT 也能够实现类似功能，不过作业需要切分到三个不同的方法中，每个方法使用不同的事务属性。总体而言，采用 UMT 方式要方便得多。

类似地，通过 UMT 的 servlet 可以创建跨多个方法调用的事务访问 EJB。要想使用 CMT 达成同样的目的，你需要向 EJB 的接口中加入一个新的元方法（metamethod），用于调用同一事务中的其他方法。

在 Java SE 应用中，实体管理器（entity manager）负责提供事务对象，而应用负责划分事务对象的边界。使用 JPA 保存股票价格到数据库的例子包含下面的代码：

```
public void run() {
    for (int i = startStock; i < numStocks; i++) {
        EntityManager em = emf.createEntityManager();
        EntityTransaction txn = em.getTransaction();
        txn.begin();
        while (!curDate.after(endDate)) {
            StockPrice sp = createRandomStock(curDate);
            if (sp != null) {
```

```
                em.persist(sp);
                for (int j = 0; j < 5; j++) {
                    StockOptionPriceImpl sop =
                        createRandomOption(sp.getSymbol, sp.getDate());
                    em.persist(sop);
                }
            }
            curDate.setTime(curDate.getTime() + msPerDay);
        }
        txn.commit();
        em.close();
    }
}
```

与 JDBC 中观察到的事务行为类似，使用 JPA 时，在事务提交的频率以及自然事务边界的定义上也存在很多效率上的取舍。这个例子中的选择事务提交时机的考量会在下一节中展开讨论。

XA 事务

JPA 实体会频繁参与到 XA 事务中；这里解释一下，XA 事务是使用了多个数据库资源，或者同时使用了数据库及其他事务资源（譬如 JMS）的事务。

跨多个不同事务资源提交事务是一种非常昂贵的操作——为了提交数据，完成这种操作的算法通常都要实现可扩展架构（eXtended Architecture，XA）标准。这是一种非常智能和复杂的操作，它要求在事务涉及的各个资源之间有多次的交换往返。

这种情况下，大多数的 Java EE 应用服务器都支持一种优化，借此最后的资源能绕过正常的 XA 协议。这种优化有多个名称，譬如最终代理优化（Last Agent Optimization，LAO）、日志最终资源（Logging Last Resource，LLR）优化、最终资源提交优化（Last Resource Commit Optimization，LRCO）、最终资源开局（Last Resource Gambit），或者其他名字。

从技术上来说，这些优化并不完全相同。尤其是，如果最终的代理能够存储 XA 日志的 JDBC 资源，LLR 和 LRCO 优化就能提供完全的 ACID[1] 兼容性。有的 LAO 实现能达到这一标准，有的目前还做不到。如果 JPA 数据库能用于存放支持 LAO 的应用服务器的事务日志，事务的处理就会显著加速，因为数据库的更新不再需要使用 XA 协议。它们同时也是 ACID 兼容的。

如果具体的 LAO 实现并未像上述那样存放事务日志，它仍然具备巨大的性能优势——不过要注意的是，如果在事务进行中包括了这样的资源，一旦发生崩溃，这些崩溃是无法自动恢复的。这时，人力的加入就变得必不可少，管理员需要查看挂起的事务以及最近提交的事务，手工地回滚一些事务以重建数据的一致性。

注 1：ACID 的全称为 Atomicity Consistency Isolation Durability，即原子性、一致性、隔离性和持久性，详见 http://www.fredosaurus.com/notes-db/transactions/acid.html。——译者注

快速小结

1. 采用 UMT 显式地管理事务的边界通常能提升应用程序的性能。
2. 默认的 Java EE 编程模型——Servlet 或者 Web Service 通过 EJB 访问 JPA 实体——很容易支持这种模式。
3. 还有另一种替代方案，即可以按照应用程序的事务需要，将 JPA 的逻辑划分到不同的方法中处理。

11.2.2　对JPA的写性能进行优化

在 JDBC 中，我们关注两类关键的性能技巧：重用预处理语句和通过批处理进行更新。

使用 JPA 有可能同时实现这两方面的优化，但是如何优化取决于使用 JPA 的具体实现；JPA 的应用程序接口并没有提供相应的调用。对于 Java SE 而言，典型的优化方法是修改应用程序的 persistence.xml 文件，以调整某个属性的设置。

> **尽量减少写入的字段**
>
> 优化数据库的写操作性能的一个比较通用的方式是只更新那些已经变化的字段。比如，在一个人力资源系统中，使我的薪水翻倍所使用的代码会访问我们员工记录中的二十个字段，但是只有一个字段（这点非常重要）要写回到数据库。
>
> JPA 的实现应该透明地完成这种优化。JPA 需要提供一种机制来记录代码中的哪些值发生了改变，这是为什么 JPA 字节码必须增强的原因之一。JPA 字节码适当增强后，用于记录我加薪记录的 SQL 语句就只会更新那个变化的字段。

譬如，使用 JPA 的 EclipseLink 应用实现时，通过在 persistence.xml 文件中添加下面的属性可以开启语句重用。

```
<property name="eclipselink.jdbc.cache-statements" value="true" />
```

注意，该选项是在 EclipseLink 实现的范围内开启语句重用的。如果 JDBC 驱动程序能够提供语句池（statement pool），通常建议在驱动程序级别开启语句缓存而不是在 JPA 的配置中进行这样的设置。

添加下面的属性可以在引用 JPA 实现中开启语句批处理（statement batching）：

```
<property name="eclipselink.jdbc.batch-writing" value="JDBC" />
<property name="eclipselink.jdbc.batch-writing.size" value="10000" />
```

JDBC 驱动程序无法自动实现语句批处理，所以该选项在所有的场景中都非常有帮助。批处理可以从两个方面进行控制：首先可以设置 `size` 属性，就如上面的例子所述。其次，应用可以定期地调用实体管理器的 `flush()` 方法，调用该方法会立即触发所有的批处理语句执行。

表 11-2 展示了使用语句重用和批处理向数据库创建和写入股票实体的效果。

表11-2：使用JPA插入128条股票记录的耗时对比

编程模式	消耗时间（秒）
不使用批处理和语句池	240
不使用批处理，使用语句池	200
使用批处理，不使用语句池	23.37
同时使用批处理和语句池	21.08

快速小结

1. JPA 应用和 JDBC 应用一样，受益于对数据库写操作调用的次数限制（有时还需要权衡是否持有事务锁）。
2. 语句缓存可以在 JPA 层面实现，也可以在 JDBC 层面实现。不过，我们应该首先使用 JDBC 层面的缓存。
3. 批量的 JPA 更新可以通过声明（在 persistence.xml 文件中）实现，也可以通过编程方式（通过调用 `flush()` 方法）实现。

11.2.3　对JPA的读性能进行优化

优化 JPA 的读操作，确定何时以及如何从数据库中读取数据是非常复杂的，远不像看起来那么简单。原因在于为了满足将来潜在的请求，JPA 会缓存数据。对提高性能而言，这通常是个不错的设计，但是它也意味着由 JPA 生成的用于读取数据的 SQL 语句，从数据量的角度看似乎并不太令人满意。数据检索针对 JPA 缓存的情况进行优化，而非针对正在进行的请求进行优化。

关于缓存的细节，我们在下一节会深入讨论。现在先看看对 JPA 进行数据库查询优化的基本方式。JPA 在三种情况下需要从数据库中读取数据，分别是：`EntityManager` 的 `find()` 方法被调用时；一个 JPA 查询执行时；代码访问一个新的实体，该实体会使用现有实体的一些关系时。就股票类而言，它属于最后一种情况，它在股票实体上调用了 `getOptions()` 方法。

调用 `find()` 方法是最直观的情况：这时只涉及一行记录，该行记录（至少一行）会从数据库中读取。这里能控制的是有多少数据会返回。JPA 可以只返回该行中的某些字段，也可以返回整行的数据，它还可以预取与正在处理的行相关的其他实体数据。这些优化都能够应用到查询上。有两种可能的优化途径：读更少的数据（因为这些数据都不需要），或者一次读取更多的数据（因为这些数据将来一定会被访问）。

1. 读取更少的数据

为了读取更少的数据，你需要指定哪些字段要延迟载入。查询实体时，被声明为延迟载入的字段会从查询载入数据的 SQL 语句中移除。如果该字段的 getter 方法被执行，这意味着需要再次查询数据库才能取得该字段的数据。

我们很少在基本类型的简单列上使用该声明，不过如果实体包含大型的 BLOB 或者 CLOB 对象，就需要考虑是否使用这种声明了。

```
@Lob
@Column(name ="IMAGEDATA")
@Basic(fetch = FetchType.LAZY)
private byte[] imageData;
```

在这个例子中，实体映射到了一个存储二进制图像数据的表中。二进制数据非常大，因此例子认为这部分数据除非有真实的需要，否则没有必要载入。不载入不需要的数据在这里带来了两个好处：查询实体时，SQL 的运行速度更快了；除此之外，它还节省了大量内存，直接减轻了垃圾回收的压力。

> **提取组（Fetch Groups）**
>
> 如果实体有些字段被定义为延迟载入（lazy load），通常它们会在需要访问时一次一个地被载入。
>
> 如果实体中有多个字段（譬如三个）都被定义为延迟载入，如果一个字段被访问，它们有可能都被访问吗？如果答案是确定的，最好一次性载入所有延迟载入的字段。
>
> 以上并不是 JPA 标准的一部分，但是大多数的 JPA 实现都允许自行定义提取组来完成这样的任务。使用提取组，我们可以指定哪些延迟载入的字段可以作为一个群组，一旦这个群组中的一个成员被访问，整个群组都会被载入。典型的情况下，我们可以定义多个相互独立的字段群组，每个群组都可以在需要的时候载入。
>
> 由于不是 JPA 的标准，所以使用提取组的代码都依赖于特定的 JPA 实现。但是如果有需要，有关的细节请查阅你使用的 JPA 实现文档。

也请注意，延迟的声明最终只是对 JPA 实现的建议。JPA 实现可以自由决定是否要对数据采用主动载入，或者延迟载入。

另一方面，可能有些数据需要提前载入——譬如，获取一个实体时，其他（相关）的实体也应该返回。这种情况被称为预取（eager fetching），它使用类似的注释：

```
@OneToMany(mappedBy="stock", fetch=FetchType.EAGER)
private Collection<StockOptionPriceImpl> optionsPrices;
```

如果实体之间的关系类型定义是 `@OneToOne` 或者 `@ManyToOne`，默认情况下这些相关实体就会被主动载入（相反的优化可以采取类似的方式：如果实体几乎不会被使用，可以将它们标记为 `FetchType.LAZY`）。

对 JPA 实现而言，这也只是一种建议，但它本质上想要表达的是，任何时候，我们在提取股票价格实体的同时，应该确保所有相关期权价格也同时被返回。有一点需要注意：通常我们认为主动方式的载入在生成的 SQL 中会使用 `JOIN`。但是典型的 JPA 供应商都没有使用这种方式：它们会执行一条 SQL 查询取得主要的对象，紧接着再执行一条或多条 SQL 命令取得其他相关的对象。如果你使用 `find()` 方法，这里就没有任何限制：如果需要使用 `JOIN` 语句，你可以自己构造查询，并在程序的查询中调用 `JOIN` 查询。

2. 在查询中使用JOIN

JPQL 不允许你指定返回对象的哪些字段。以下面的 JPQL 查询为例：

```
Query q = em.createQuery("SELECT s FROM StockPriceImpl s");
```

这个查询产生的是下面的这条 SQL 语句：

```
SELECT <enumerated list of non-LAZY fields> FROM StockPriceTable
```

如果你希望在生成的 SQL 中返回更少的字段，没有其他的办法，只能将它们标记为延迟载入类型。同样地，对标记为 lazy 的字段，没有其他选项可以在查询中返回它们的内容。

如果实体间有关系，这些实体可以在 JPQL 中使用显式的联合查询，一次性地返回初始实体以及与它相关的其他实体。以前文的股票实体为例，我们可以构造下面的查询：

```
Query q = em.createQuery("SELECT s FROM StockOptionImpl s " +
                         "JOIN FETCH s.optionsPrices");
```

这条 JPQL 会生成类似下面的 SQL 语句：

```
    SELECT t1.<fields>, t0.<fields> FROM StockOptionPrice t0, StockPrice t1
    WHERE ((t0.SYMBOL = t1.SYMBOL) AND (t0.PRICEDATE = t1.PRICEDATE))
```

联合查询（JOIN FETCH）的其他机制

很多 JPA 的提供商允许通过在查询上设置查询提示（Query Hint）指定进行联合查询。譬如，在 EclipseLink 中，下面的代码会生成 JOIN 查询：

```
Query q = em.createQuery("SELECT s FROM StockOptionImpl s");
q.setQueryHint("eclipselink.join-fetch", "s.optionsPrices");
```

有的 JPA 提供商还提供了特殊的 @JoinFetch 注释，可用于这种关系的定义。

不同的 JPA 供应商生成的具体 SQL 语句可能略有不同（本例使用的是 EclipseLink），但是通用的流程都大同小异。

对实体关系（Entity Relationship）而言，无论它们被注解为主动载入还是延迟载入，都可以使用联合查询。如果 join 应用于延迟载入关系的实体，且注释为延迟载入的实体满足查询条件，这部分实体也会从数据库中返回，且这部分实体在将来使用时，不需要再次访问数据库。

联合查询返回的所有数据都被使用时，其带来的性能提升效果最显著。然而，联合查询还会以一些难以预期的方式和 JPA 缓存进行交互。这种情况在介绍缓存的一节将会举例阐述；正式使用联合查询编写自定义的查询之前，请确保你已经完全理解各种可能的后果。

3. 批处理和查询

JPA 查询的处理和 JDBC 查询一样，都会产生结果集：JPA 的实现提供了多种选择，可以一次取得所有的数据，或者随着应用程序遍历整个查询结果，每次取得一条记录，或者每次读取若干个结果（类似于 JDBC 提取大小那样工作）。

标准并未规定这些情况如何处理，但是 JPA 的提供商们大都有自己的专有机制设置提取大小。譬如，EclipseLink 中对查询设置了 hint 来指定提取大小，如下所示：

```
q.setHint("eclipselink.JDBC_FETCH_SIZE", "100000");
```

与此相反，Hibernate 则提供了自定义的 `@BatchSize` 声明来设定提取大小。

如果有超大量的数据正在处理，代码可能需要对返回的列表进行分页处理。这与数据如何在网页上呈现给用户有天然的联系：刚开始时，数据的一个子集呈现在页面上（譬如 100 行记录），随着点击“向后”、“向前”的页面链接，我们就能通过页面浏览所有的数据，这些都是通过对查询设置返回区间实现的。

```
Query q = em.createNamedQuery("selectAll");
query.setFirstResult(101);
query.setMaxResults(100);
List<? implements StockPrice>  = q.getResultList();
```

这个查询返回的是用于呈现在 Web 应用第二页的列表数据：101 号到 200 号条目。相对于提取 200 行数据，之后再丢弃前 100 行而言，只提取这个区间的数据更加高效。

注意，这个例子使用了命名查询（通过 `createNamedQuery()` 方法），没有使用即时查询（即 `createQuery()` 方法）。在很多 JPA 实现中，命名查询的运行速度都更快：JPA 的实现几乎都结合语句缓存池使用了带绑定参数的预处理语句。没有任何规定禁止 JPA 实现使用类似于匿名或即时查询应用的逻辑，只不过这种情况实现的难度会更大，而 JPA 的实现可能仅仅是每次创建一个新的语句（即一个 `Statement` 对象）。

快速小结

1. JPA 会进行多种优化，以限制（或增加）一次读取操作所返回的数据量。
2. JPA 实体中，如果一个大型字段（譬如 BLOB 类型的字段）很少被使用，就应该延迟载入。
3. JPA 实体之间存在关系时，相关的数据可以主动载入或者延迟载入，具体的选择取决于应用程序的需要。
4. 采用主动载入关系时，可以使用命名查询生成一个使用 `JOIN` 的 SQL 语句。应注意的是，这会影响 JPA 缓存，因此并不总是最好的主意（下一节会讨论它的影响）。
5. 使用命名查询读取数据比普通的查询要快很多，因为 JPA 实现为命名查询构造 `PreparedStatement` 更容易。

11.2.4 JPA缓存

Java 与性能相关的最经典的用例之一是它提供了一种机制可以充当中间层，缓存后端数据库返回的数据。从架构的角度看，Java 层完成了几个重要的功能（譬如，避免客户端直接访问数据库）。从性能的角度出发，在 Java 层缓存频繁使用的数据能极大地加速客户端的响应时间，从而改善用户体验。

JPA 从设计之初就秉持了这样的架构设计。在 JPA 中包含了两类缓存。每个实体管理器实例都有自己的缓存：它会在本地缓存事务中取得的数据。与此同时，它还会在本地缓存事务中写入的数据；只有在事务提交时，这些缓存的数据才会发送给数据库。一个程序可能包含多个不同的实体管理器实例，每个管理器执行着不同的事务，每个也都有自己的本地缓存（尤其是，插入到 Java EE 应用中的实体管理器都是相互独立的实例）。

实体管理器提交事务时，本地缓存中的所有数据会合并到全局缓存中。全局缓存对应用程序的所有实体管理器而言是共享的。全局缓存也被称为二级缓存（L2 Cache）或者是二层缓存（Second-Level Cache）；而实体管理器中的缓存被称为一级缓存（L1 Cache）或者是一层缓存（First-Level Cache）。

实体管理器的事务缓存基本上不需要进行调优，且在所有的 JPA 实现中，L1 缓存默认都是开启的。L2 缓存则稍有不同：大多数 JPA 实现只提供了一个缓存，而不是默认开启所有的缓存（譬如，Hibernate 默认并未开启所有的缓存，不过 EclipseLink 默认就启动了所有的缓存）。一旦开启 L2 缓存，它的调优和使用就会极大地影响应用的性能。

JPA 缓存只在通过它们的主键访问实体时有效，即通过调用 `find()` 方法返回对象，或者通过访问相关实体（或者主动载入）得到对象。实体管理器尝试通过它的主键或者关系映射查找对象时，首先会在 L2 缓存中查找，如果找到满足条件的对象就直接返回，从而节省了访问数据库的开销。

一般通过查询返回的对象不会在 L2 缓存中保存。有些 JPA 实现会提供自己独特的机制对查询的结果进行缓存，但是这些结果只有在几乎完全相同的查询再次执行时才能重用。即使 JPA 的实现支持查询缓存，实体自身也不会保存在 L2 缓存中，因此无法在之后调用 `find()` 方法时返回。

L2 缓存、查询以及对象的载入之间关系紧密，多种方式都会影响应用程序的性能。为了说明这些关系，我们使用下面的循环代码进行例证：

```
EntityManager em = emf.createEntityManager();
Query q = em.createNamedQuery(queryName);
List<StockPrice> l = q.getResultList(); ❶
for (StockPrice sp : l) {
    …… 处理sp ……
    if (processOptions) {
        Collection<? extends StockOptionPrice> options = sp.getOptions(); ❷
        for (StockOptionPrice sop : options) {
            …… 处理sop ……
        }
    }
}
em.close();
```

❶ SQL Call Site 1

❷ SQL Call Site 2

由于 L2 缓存的存在，循环的第一次执行与之后的执行（通常会更快）的路径是不一样的。具体性能上的差异取决于查询的细节以及实体关系。接下来的几个子节会详细探讨这些结果的差异。

这个例子中的差异主要源于 JPA 配置的不同，但也包括一些测试的运行并未遍历 `Stock` 和 `StockOptions` 类之间的关系。这些没有遍历关系的测试中，`processOptions` 在循环中的值为 `false`；因此实际在使用的只有 `StockPrice` 对象。

1. 默认缓存（延迟载入）

示例代码中，股票价格通过命名查询载入。默认情况下都会使用这种简单的查询载入股票数据：

```
@NamedQuery(name="findAll",
    query="SELECT s FROM StockPriceImpl s ORDER BY s.id.symbol")
```

`StockPrice` 类与 `StockOptionPrice` 之间通过 `optionPrices` 实例变量有 `@OneToMany` 的关系：

```
@OneToMany(mappedBy="stock")
private Collection<StockOptionPrice> optionsPrices;
```

`@OneToMany` 关系默认采用延迟载入机制。表 11-3 展示了执行这个循环所消耗的时间。

表11-3：读取128支股票数据消耗的时间（默认配置）

测试用例	首次执行	后续执行
延迟载入	61.9 秒（33 409 次 SQL 调用）	3.2 秒（1 次 SQL 调用）
延迟载入，不遍历	5.6 秒（1 次 SQL 调用）	2.8 秒（1 次 SQL 调用）

示例循环首次运行这种场景（读取 128 支股票一年的数据）时，JPA 代码会调用 `executeQuery()` 方法执行一条 SQL 语句。该语句会执行代码列表中第一部分的 SQL 调用（SQL Call Site 1）。

随着代码遍历该股票，读取期权价格的集合，JPA 会执行 SQL 语句提取与特定实体相关的所有期权信息（即该语句会一次性地返回某支股票 / 日期的整个集合）。这就是第二部分的 SQL 语句，执行过程中它会生成 33 408 个单独 `SELECT` 语句（261 天 × 128 支股票）。

首次执行循环时，这个例子耗时 62 秒钟。第二次执行时，只花费了 3.2 秒。这是因为第二次循环运行时，仅执行了命名查询。通过关系提取的实体还保持在 L2 缓存内，所以这种情况无需访问数据库。（前面提过，L2 缓存只作用于通过关系加载或者 `find()` 操作的实体。所以我们才能在 L2 缓存中找到股票期权实体，但是股价就不行——因为股价是由查询载入的，在 L2 缓存内不存在，所以必须重新载入。）

表 11-3 的第二行代表的是不依次遍历访问每个期权的情况（譬如，`processOptions` 变量值为 `false`）。这种情况下，代码的运行速度会大大提升：完成循环的第一次迭代仅耗时 5.6 秒，紧接着的迭代耗时 2.8 秒。（这两个例子的性能差异源于编译器的预热。虽然我们很难察觉到，但这个预热在第一个例子中也存在。）

2. 缓存和主动载入（Eager Loading）

接下来的两个实验中，为了主动载入期权价格，股票价格与期权价格之间的关系进行了重新定义。

所有数据都被使用时（譬如表 11-3 和表 11-4 中第一行的情况，主动载入与延迟载入的效果几乎是一样的。但是如果载入的相关数据实际并未使用（如每张表中第二行的情况），采用延迟载入相关数据的方式能够节省一些时间——尤其循环第一次执行时。若在紧接着的执行中采用这种方式，循环就无法再节省时间了，因为主动载入的代码在之后的执行过程中不会重新载入，而是直接从 L2 缓存中读取。

表11-4：读取128支股票所消耗的时间（采用主动载入）

测试用例	首次执行	后续执行
主动载入	60.2 秒（33 409 次 SQL 调用）	3.1 秒（1 次 SQL 调用）
主动载入，不遍历	60.2 秒（33 409 次 SQL 调用）	2.8 秒（1 次 SQL 调用）

主动载入相关数据

无论相关数据是通过延迟方式载入还是通过主动方式载入，这个循环都会执行 33 408 个 SELECT 语句以取得相应的股票期权（正如上一节所提到的，默认不会使用 JOIN）。

这种情况下，主动载入和延迟载入的区别是什么时候这些 SQL 语句会执行。如果指定的关系是主动载入，查询运行时结果集立即会被处理（在 getResultList() 方法调用内进行）。JPA 框架会查看该调用返回的每个实体，执行相关的 SQL 语句，提取关联实体。所有这些 SQL 语句的执行都发生在 SQL 调用区 1（SQL Call Site 1）——采用主动载入时，没有任何 SQL 语句会在 SQL 调用区 2（SQL Call Site 2）内运行。

如果指定的关系是延迟载入，在 SQL 调用区 1 内（使用命名查询）只有股票价格会载入。股票的期权价格会在 SQL 调用区 2 进行关系遍历时载入。这个循环会运行 33 408 次，因此会触发 33 408 次 SQL 调用。

无论 SQL 在什么时候执行，SQL 语句的总数是固定的——我们假定延迟载入的例子中所有的数据都会被使用。

3. 联合查询和缓存

正如前一节中所讨论到的，我们可以显式地使用 JOIN 语句编写查询：

```
@NamedQuery(name="findAll",
    query="SELECT s FROM StockPriceEagerLazyImpl s " +
    "JOIN FETCH s.optionsPrices ORDER BY s.id.symbol")
```

使用命名查询（结合全遍历）的数据如表 11-5 所示：

表11-5：读取128支股票的耗时（使用联合查询）

测试用例	首次执行	后续执行
默认配置	61.9 秒（33 409 次 SQL 调用）	3.2 秒（1 次 SQL 调用）
联合查询	17.9 秒（1 次 SQL 调用）	11.4 秒（1 次 SQL 调用）
带查询缓存的联合查询	17.9 秒（1 次 SQL 调用）	1.1 秒（0 次 SQL 调用）

首个循环中使用联合查询得到了极大的性能提升：完成本次循环仅耗时 17.9 秒。这是执行一个 SQL 请求的结果，而不是 33 409 次 SQL 查询的结果。

不幸的是，下一次代码执行还需要再次运行那条同样的 SQL 语句，因为查询的结果没在 L2 缓存内保存。这个例子接下来的执行耗时 11.4 秒——这是由于执行的 SQL 语句包含 JOIN 语句，返回的数据超过 200 000 行记录。

如果 JPA 的提供商支持查询缓存，这个场景下使用该机制无疑能极大地改善程序的性能。

如果代码第二次执行时不需要再次运行 SQL 语句，接下来的执行耗时就只需 1.1 秒。注意，查询缓存只有在每次查询运行时使用的参数都完全相同时才工作。

4. 避免查询

如果实体不需要通过查询取得，那么经过初始的预热阶段后，所有的实体都可以通过 L2 缓存访问。L2 缓存可以通过载入所有的实体预热，所以我们在之前例子代码的基础上稍加修改得到了下面的代码：

```
EntityManager em = emf.createEntityManager();
ArrayList<String> allSymbols = …… 所有有效的股票 ……;
ArrayList<Date> allDates = …… 所有有效的日期……;
for (String symbol : allSymbols) {
    for (Date date = allDates) {
        StockPrice sp =
            em.find(StockPriceImpl.class, new StockPricePK(symbol, date);
        …… 处理sp ……
        if (processOptions) {
            Collection<? extends StockOptionPrice> options = sp.getOptions();
            …… 处理选项 ……
        }
    }
}
```

这段代码的运行结果如表 11-6 所示。

表11-6：读取128支股票数据的耗时（使用L2缓存）

测试用例	首次执行	后续执行
默认配置	61.9 秒（33 409 次 SQL 调用）	3.2 秒（1 次 SQL 调用）
不作查询	100.5 秒（66 816 次 SQL 调用）	1.19 秒（0 次 SQL 调用）

首次执行这个循环需要 66 816 个 SQL 语句：`find()` 方法要执行 33 408 次 SQL 调用，`getOptions()` 方法又执行了 33 408 次 SQL 调用。所以，优化之后这段代码的速度提升了很多，因为现在所有的实体都保持在 L2 缓存内，不再需要执行任何的 SQL 语句了。

> **测试预热**
>
> Java 性能测试——尤其是基准测试，通常都有个预热阶段。正如我们在第四章中讨论的，预热能帮助编译器编译出优化的代码。
>
> 这是另一个证明预热阶段极其有益的例子。JPA 应用的预热阶段中，最常使用的实体会被载入到 L2 缓存中。通过对不同测试期间的度量，我们看到随着实体第一次载入，应用的性能发生了显著的变化。这一点在上一个例子中尤其突出，因为优化后不再需要使用查询载入实体了。

前面提过，样本数据库中包含每个日期及股票组合对应的五个期权价格，或者 128 支股票超过一年的数据，总计 167 040 笔期权的价格。通过关联访问某支股票在某个日期的五个股票期权，这些数据会一次性地返回。这就是为什么载入所有的期权价格需要执行 33 408 条 SQL 语句。虽然运行的 SQL 语句会返回多行数据，JPA 自身仍能缓存这些返回的实

体——这和单执行一条查询的情况略有不同。如果 L2 缓存是通过遍历实体的方式构建的，就不要用循环的方式访问相关实体——通过关联可以非常容易地访问相关的实体。

随着代码优化，你必须考虑缓存的影响（尤其是 L2 缓存的影响）。即使你认为自己编写的 SQL 比 JPA 自动生成的更优（因此会使用更复杂的命名查询），也应权衡缓存发挥作用时，这样的代码是否还有意义。虽然使用简单的命名查询能快速地载入数据，也应考虑，如果这些实体是通过调用 `find()` 方法载入 L2 缓存，长期来看会造成什么影响。

5. 调整JPA缓存的大小

正如所有利用对象重用的机制，JPA 缓存也有同样的问题，它可能会对性能产生潜在的冲击：如果缓存消耗了过多内存，垃圾回收就会面临巨大的压力。出现这种情况，你往往需要调整缓存大小，或者控制哪些实体可以继续保持在缓存内。不幸的是，这方面并没有标准选项，因此你必须依据使用的 JPA 提供商，针对性地进行调优。

通常 JPA 实现都提供了选项来对缓存大小进行设置，要么是全局的设置，要么是针对每个实体的设置。显然，后者的适应性更广、更灵活，不过它也要求为确定每个实体的最优大小做更多的工作。一个替代方案是使用软引用或者弱引用，作为 JPA 实现的 L2 缓存。譬如，对 EclipseLink 而言，依据不同的弱引用或者软引用组合提供五种不同的缓存类型（包括额外不推荐使用的类型）。虽然这种方式与为每个实体都定义最优大小比起来相对更容易一些，不过它也需要做一些计划：尤其是，我们在第 7 章中介绍过，弱引用在所有垃圾回收中都会被清理，因此不适合用作缓存的对象。

如果缓存是基于软引用或者弱引用的，则应用的性能也受制于堆的使用情况。本节所有的例子使用的堆都比较大，因此即使缓存应用中的 200 448 个实体对象，也不会给垃圾收集带来任何问题。为了更优的性能，如果 JPA L2 缓存很大，则对堆进行调优是非常重要的。

快速小结

1. JPA 的 L2 缓存会自动对应用的实体进行缓存。
2. L2 缓存不会对查询返回的实体进行缓存。长期来看，这种方式有利于避免查询。
3. 除非使用的 JPA 实现支持查询缓存，否则使用 `JOIN` 查询的效果通常会对程序的性能造成负面的效果，因为这种操作没有充分利用 L2 缓存。

11.2.5 JPA的只读实体

JPA 规范并未直接定义只读实体，但是很多 JPA 供应商提供了该功能。通常情况下，只读实体比（默认的）读写实体性能更好，因为 JPA 实现很明确地知道它不需要跟踪实体状态，不必在事务中注册实体，也不必对实体上锁，等等。Java EE 的容器通常都支持只读实体，无论使用的是哪种 JPA 实现。这种情况下，应用服务器需要确保实体的访问使用非事务型（Non-Transactional）JDBC 连接。一般情况下，这种方式能带来显著的性能提升。

JPA 规范中定义了如何在 Java EE 容器中支持只读实体的事务：可以在事务之外运行一个通过 `@TransactionAttributeType.SUPPORTS` 注释的业务方法（假定该方法调用的同时没有事务在运行）。

在这种情况下，该方法访问的实体必须是只读的，因为它们不是事务的一部分。然而，如果方法是从该事务的某个方法中调用的，实体就会成为事务的一部分。

11.3 小结

合理调优访问数据库的 JDBC 和 JPA 是影响中间层应用性能最重要的因素之一。请牢记下面的最佳实践。

- 通过合理配置 JDBC 或者 JPA，尽可能地实现批量读取和写入。
- 优化应用使用的 SQL 语句。对于 JDBC 应用，这都是一些基本、标准的 SQL 命令。对 JPA 应用，你还需要考虑 L2 缓存的影响。
- 尽量减少锁的使用。如果数据不大容易发生冲突，推荐使用乐观锁（Optimistic Locking）；如果数据经常发生冲突，推荐使用悲观锁（Pessimistic Locking）。
- 请务必使用预处理语句池（Prepared Statement Pool）。
- 请务必合理设置连接池的大小。
- 合理地设置事务的范围：由于锁在整个事务期间都需要保持，所以在不影响应用程序扩展性的前提下，尽可能把事务的范围设置得大一些。

第 12 章

Java SE API技巧

在某些 Java SE API 的实现中，存在一些会影响性能的怪异行为，本章将对其进行探讨。JDK 中有很多这样的实现细节，我经常会发现由其导致的性能问题（即便在我自己的代码中，也会存在）。

12.1 缓冲式I/O

在我于 2000 年加入 Java 性能团队（Java Performance Group）时，我的老板刚出版了一本探讨 Java 性能的书，这是该领域有史以来的第一本书。缓冲式 I/O 是当时最热的话题之一。14 年过去了，我原以为这是老生常谈的话题，准备将其去掉。然而就在我着手编写本书大纲的那周，在两个毫无关联的项目中，我发现了一个问题：非缓冲式 I/O 对性能影响很大。又是几个月之后，在为本书准备例子时，我抓耳挠腮地想，为什么我的“优化”会如此之慢呢？然后我意识到：真蠢，忘记正确地缓冲 I/O 了。

下面就来谈一下缓冲式 I/O 的性能。`InputStream.read()` 和 `OutputStream.write()` 方法操作的是一个字符。由于所访问资源不同，这些方法有可能非常慢。而在 `fileInputStream` 上调用 `read()` 方法，更是慢得难以形容：每次调用该方法，都要进入内核，去取一个字节的数据。在大多数操作系统上，内核都会缓冲 I/O，因此，很幸运，该场景不会在每次调用 `read()` 方法时触发一次磁盘读取操作。但是这种缓冲保存在内核中，而非应用中，这就意味着每次读取一个字节时，每个方法调用还是会涉及一次代价高昂的系统调用。

写数据也是如此：使用 `write()` 方法向 `fileOutputStream` 发送一个字节，也需要一次系统调用，将该字节存储到内核缓冲区中。最后（当文件关闭或刷新时），内核会把缓冲区中的内容写入磁盘。

对于使用二进制数据的文件 I/O，记得使用一个 `BufferedInputStream` 或 `BufferedOutputStream`

来包装底层的文件流。对于使用字符（字符串）数据的文件 I/O，记得使用一个 `BufferedReader` 或 `BufferedWriter` 来包装底层的流。

在探讨文件 I/O 时，这一性能问题很好理解，不过它几乎存在于所有类型的 I/O 中。从 Socket 返回的流（通过 `getInputStream()` 或 `getOutputStream()`）是以同样的方式运作的，在 Socket 上每次读写一个字节的 I/O 操作相当慢。所以同样要记得正确地使用一个缓冲过滤器流来包装一下。

在使用 `ByteArrayInputStream` 和 `ByteArrayOutputStream` 类时，微妙的问题更多。首先，这些类基本上就是大的内存缓冲区。在很多情况下，用缓冲管理器流包装它们，意味着数据会被复制两次：一次是缓冲在过滤器流中，一次是缓冲在 `ByteArrayInputStream` 中（输出流的情况相反）。除非还设计了其他流，否则这种情况下应该避免缓冲式 I/O。

当涉及其他过滤器流时是否要用缓冲，这个问题就更复杂了。有种常见的情况是使用这些流来序列化或反序列化数据。比如，第 10 章就探讨了显式地管理类的数据序列化时的各种性能得失。在那一章中，有一个简化版的 `writeObject()` 方法，如下所示：

```
private void writeObject(ObjectOutputStream out) throws IOException {
    if (prices == null) {
        makePrices();
    }
    out.defaultWriteObject();
}

protected void makePrices() throws IOException {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    ObjectOutputStream oos = new ObjectOutputStream(baos);
    oos.writeObject(prices);
    oos.close();
}
```

在这种情况下，如果将 `baos` 流包装在一个 `BufferedOutputStream` 中，因为多了一次数据复制，所以会有性能损失。

在这个例子中，将 `prices` 数组中的数据压缩一下，效率会更高，而代码就变成了下面这样：

```
private void writeObject(ObjectOutputStream out) throws IOException {
    if (prices == null) {
        makeZippedPrices();
    }
    out.defaultWriteObject();
}

protected void makeZippedPrices() throws IOException {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    GZIPOutputStream zip = new GZIPOutputStream(baos);
    BufferedOutputStream bos = new BufferedOutputStream(zip);
    ObjectOutputStream oos = new ObjectOutputStream(bos);
    oos.writeObject(prices);
    oos.close();
    zip.close();
}
```

现在，缓冲数据流就是必要的了，因为 GZIPOutputStream 处理一块数据比处理一个字节的数据更高效。在上面这两种情况下，ObjectOutputStream 都会将单字节数据发送给下一个流。如果下一个流就是最终目的地，比如 ByteArrayOutputStream 这种情况，则无需缓冲。而如果中间还有另一个过滤器流（比如这个例子中的 GZIPOutputStream），则缓冲往往是必要的。

到底何时需要在两个不同的流中间插入一个缓冲流，并没有一个统一的规则。这最终取决于所用流的类型，但是在多数情况下，操作一块数据（来自缓冲的流）通常要好于操作一系列单个字节（来自 ObjectOutputStream）。

同样的情况也适用于输入流。举个具体的例子，GZIPInputStream 操作一块字节数据更高效；一般情况下，对于插入在 ObjectInputStream 和原始的字节数据源之间的流，如果配合一块数据，其表现会更好。

注意，这种情况特别适用于编解码器流。当在字节和字符之间转换时，操作尽可能大的一段数据，性能最佳。如果提供给编解码器的是单个的字节或字符，性能会很差。

郑重声明，我在编写压缩这个例子时犯过的错误就是没有缓冲 gzip 流。如表 12-1 中的数据所示，这个错误代价很高。

表12-1：压缩情况下，序列化/反序列化10 000个对象所需时间

操作	序列化时间（秒）	反序列化时间（秒）
无缓冲压缩 / 解压缩	60.3	79.3
带缓存压缩 / 解压缩	26.8	12.7

没有正确地缓冲 I/O，性能差距多达 6 倍。

快速小结

1. 围绕缓冲式 I/O 有一些很常见的问题，这是由简单输入输出流类的默认实现引发的。
2. 文件和 Socket 的 I/O 必须正确地缓冲，对于像压缩和字符串编解码等内部操作，也是如此。

12.2　类加载

对于任何尝试优化程序启动或优化新代码在动态系统中的部署（比如向 Java EE 应用服务器中部署一个新应用，或者是在浏览器中加载一个 Applet）的人而言，类加载的性能都让人头疼。

原因是多方面的。最主要的一点是，类数据（也就是 Java 字节码）通常无法快速访问到。它必须从磁盘或者网络上加载过来，必须能在 classpath 下的某个 JAR 文件中找到，还必须能在某个类加载器中找到。对此有一些改进方案，比如，Java WebStart 会将从网络读取的类数据写入一个隐藏目录，这样当下次启动同一应用时，就可以从本地磁盘读取数据，而不再需要从网络读取，速度得以提升。在打包应用时，减少所生成的 JAR 文件数，也能提升类加载的性能。

在复杂环境中，提升速度的明显方式之一就是将类加载并行化。以一个典型的应用服务器为例：在启动时，它可能需要初始化多个应用，其中每个应用都使用了自己的类加载器。假设有多个 CPU 可供大部分应用服务器使用，并行化应该有明显优势。

不过有两个因素会影响其可伸缩性。第一，类数据很可能保存在同一个磁盘上，因此如果有两个类加载器并发运行，它们会向同一设备发出读请求。尽管操作系统擅于处理这种情况：它们可以随着磁盘旋转，分割读操作，并抓取字节数据，但是此时磁盘仍然有很大机会成为瓶颈。

在 Java 7 之前，`ClassLoader` 类本身的设计一直存在一个比较大的问题。如图 12-1 所示，Java 的类加载器存在于一个层次结构中，这是某个 Java EE 容器中类加载器的理想情况。当运行在第一个 Servlet 应用中的类加载器需要某个类时，请求会流向第一个 Web 应用类加载器（App Classloader），但是这个类加载器会将请求委派给其父类加载器：系统类加载器（System Classloader）。它是与 classpath 关联的类加载器，负责加载 Java EE 相关的类（比如 Java Server Faces，即 JSF 接口）以及这些类在该容器中的实现。系统类加载器也会将一些加载工作委派给其父类加载器，即启动类加载器（Bootstrap Classloader），它负责加载核心 JDK 类。

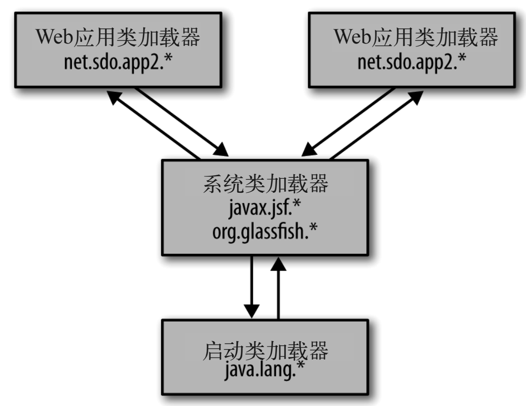


图 12-1：多个类加载器的理想结构

最后的结果是，当请求加载一个类时，启动类加载器是尝试去找这个类的第一段代码[1]；之后是系统类加载器（classpath），再找不到则诉诸应用类加载器。从功能角度看，这是说得通的：`java.lang.String` 类的字节码必须由启动类加载器加载，而不能是由层次结构中的其他某个类加载器有意或无意加载的别的实现。

在 Java 7 之前，类加载器存在的问题是，用于加载类的方法是同步的：在某个时刻，只有一个类加载器可以将任务委派给系统类加载器。这对使用多个类加载器实现并行化有着极大限

注 1：Java 的类加载器使用了双亲委派模型，一个类加载器在收到加载类的请求后，首先会把这个请求委派给父类加载器去完成，每个层次的类加载器都如此处理，一直到启动类加载器；如果启动类加载器无法完成加载请求，才会沿这个路径返回，找到合适的类加载器。——译者注

制，因为每个类加载器都要等待，只有轮到它时，才能访问系统类加载器和启动类加载器。Java 7 利用一组基于类名的锁解决了这种状况。 现在，如果有两个类加载器在寻找同一个类，它们仍然会争用某个锁，但是类层次结构中寻找其他类的类加载器可以并行执行。

如果使用了诸如 URLClassLoader 等 Java 提供的类加载器，也能体会到 Java 7 带来的这一好处。在 Java 6 中，如果其他的类加载器以 URLClassLoader 为父类加载器，它会成为并行操作的同步瓶颈；而在 Java 7 中，类加载器可以并行使用。Java 7 所提供的类加载器是可以并行的（parallel-capable）。

自定义的类加载器默认是不支持并行的。如果希望自己的类加载器也能并行使用，必须采取一些措施。措施总共分为两步。

首先，确保类加载器的层次结构中没有任何回环。回环并不多见。如果存在回环，代码会很难维护，因为在某一时刻，某个类加载器必须直接满足请求，而不能将请求传给其父类加载器（否则委派就成了无限循环）。因此，对于一组存在回环的类加载器而言，尽管在技术上支持并行是可能的，但是过程会非常复杂（要在本已非常复杂的代码上实现）。因为编写高性能 Java 代码的一个规则是采用惯用的方法，以及编写便于编译器优化的简单代码，所以我们不推荐使用存在回环的类加载器层次结构。

第二，在定义加载器类时，在静态初始化部分将其注册为可以并行的：

```
public class MyCustomClassLoader extends SecureClassLoader {

    static {
        registerAsParallelCapable();
    }

    ....
}
```

这个调用必须放在每个具体的类加载器实现之内。SecureClassLoader 本身是可以并行的，但是其子类并不会自动具备这种能力。如果在我们的代码内还有一个类继承了 MyCustomClassLoader，那个类也必须自己注册为支持并行的。

对于大部分类加载器而言，只需要这两步。在编写类加载器时，建议重写 findClass() 方法。如果自定义的类加载器重写的是 loadClass() 方法，而非 findClass() 方法，则一定要确保在每个类加载器实例内，对于每个类名，defineClass() 方法只调用一次。

在涉及围绕锁的可伸缩性时，和与此有关的所有性能问题一样，该优化的性能净收益还与代码被锁住多久有关。举一个简单的例子，考虑如下代码：

```
URL url = new File(args[0]).toURL();
URLClassLoader ucl = new URLClassLoader(url);
for (String className : classNames) {
    ucl.loadClass(className);
}
```

在命令行中，一个 JAR 文件的名字被作为 args 的第一个元素传入了，这里自定义的类加载器会在这个 JAR 文件中查找。它会遍历由类名组成的数组（在其他地方定义），并从这个 JAR 文件中加载每个类。

其父类加载器就是系统类加载器（查找 classpath）。当有两个或多个线程并发执行这个循环时，因为它们会把类的查找委派给系统类加载器，所以两个线程会彼此等待。表 12-2 列出了当系统的 classpath 为空时该循环的性能。

表12-2：并发加载类的时间（classpath为空）

线程数	在JDK 7中用的时间（秒）	在JDK 6中用的时间（秒）
1	30.353	27.696
2	34.811	31.409
4	48.106	72.208
8	117.34	184.45

这是有 1500 个类的类名列表循环 100 次所用的时间。这里可以得出几个有趣的结论。首先，JDK 7 中的代码更为复杂（以支持并行加载），因此会在最简单的情况下引入一点点性能损失——这是越简单的代码跑得越快这一原则的一个例证。即使在两个线程的情况下，JDK 7 的新模型还是要稍微慢点，因为代码在父类加载器中几乎没花时间：被锁在父类加载器上所花的时间，远远不及在程序其他地方所花的时间多。

当有 4 个线程时，情况就不一样了。首先，在有 4 个 CPU 的机器上，这 4 个线程会与其他进程（尤其像正在显示用到了 Flash 的页面的浏览器，它会占用一个 CPU 的 40%）争用 CPU 周期。因此即使在 JDK 7 中，伸缩也不是线性的。但至少可伸缩性有了提高：在 JDK 6 中，围绕父类加载器的竞争非常严重。

产生这种竞争有两个原因。首先，争用 CPU 实际上会增加类加载器锁的持有时间；其次，争用锁的线程数也成了原来的 2 倍。

增加系统 classpath 的长度也会极大增加父类加载器锁的持有时间。表 12-3 重复了这个实验，classpath 下有 266 个条目（GlassFish 发行版中的 JAR 文件数）。（GlassFish 不会简单地把这些文件都放到一个类加载器中；之所以选择它，只是方便举例而已。）

表12-3：并发加载类的时间（classpath较长的情况）

线程数	在JDK 7中用的时间（秒）	在JDK 6中用的时间（秒）
1	98.146	92.129
2	111.16	316.01
4	150.98	708.24
8	287.97	1461.5

现在，即便只有两个线程，竞争也非常严重：没有支持并行的类加载器，加载这些类的时间是原来的 3 倍。如果是在压力已经很大的系统中，可伸缩性就更糟糕了。最后，性能会慢 7 倍。

这里有一个有趣的取舍：是采用更复杂的代码，稍微牺牲一下单线程情况下的性能，还是针对其他情况做优化——特别是像上面例子中的情况，两种选择的性能差距非常大。这种性能取舍时常会遇到，在这种情况下，JDK 团队将第 2 种选择当作了默认情况。作为一个平台，同时提供这两种选择是个不错的主意（即使默认的只能有一个）。因此，在 JDK 7

中，要想获得 JDK 6 的这种行为，可以使用 -XX:+AlwaysLockClassLoader 标志（它默认为 false）开启。如果启动周期较长，而且没有并发的线程会从不同的类加载中加载类，这种情况使用该标志可能稍微有好处。

快速小结

1. 在存在多个类加载器的复杂应用（特别是应用服务器）中，让这些类加载器支持并行，可以解决系统类加载器或者启动类加载器上的瓶颈问题。
2. 如果应用是在单线程内，则通过一个类加载器加载很多类，关掉 Java 7 支持并行的特性可能会有好处。

12.3 随机数

Java 7 提供了 3 个标准的随机数生成器类：java.util.Random、java.util.concurrent.ThreadLocalRandom 以及 java.security.SecureRandom。这三个类在性能方面差距很大。

Random 和 ThreadLocalRandom 两个类的差别是，Random 类的主要操作（nextGaussian()）是同步的。任何要获取随机值的方法都会用到这个方法，所以不管如何使用该随机数生成器，都会存在锁竞争：如果两个线程同时使用同一随机数生成器，那一个线程要等待另一个先完成其操作。之所以会使用 ThreadLocalRandom，原因就在于此：每个线程都有自己的随机数生成器，Random 类的同步就不是问题了。（正如第 7 章所讨论的，因为创建对象成本很高，而 ThreadLocalRandom 类会重用对象，所以有很大的性能优势。）

SecureRandom 类与上面介绍的两个类的区别是，所用的算法不同。Random 类（以及继承它而来的 ThreadLocalRandom）实现了一个典型的伪随机数算法。尽管那些算法非常复杂，但到底是确定性的。如果知道初始种子，很容易确定该引擎将生成的数字的精确序列。这意味着，黑客能够从特定的生成器看到数字序列，也就能够指出下一个数字是什么。尽管好的伪随机数生成器可以生成看上去真正随机的数字序列（甚至符合随机性的概率期望），但这仍然不是真正的随机。

而另一方面，SecureRandom 类使用一个系统接口来获得随机数。数据生成方式与所用的操作系统有关，不过一般而言，这类源[2]提供了基于真正随机事件（比如鼠标移动时）的数据。这就是所谓的基于熵的随机性，比依赖随机数的操作更安全。SSL 熵是这类操作中最广为人知的例子：加密所用的随机数不可能通过基于熵的源来确定。（即便在算法中使用了 SecureRandom 随机数生成器，还是有其他方式可以攻破数据的加密算法。）

遗憾的是，计算机生成的熵的数量是有限的，所以要从一个安全随机数生成器获得大量的随机数，需要很长时间。调用 SecureRandom 类的 nextRandom() 方法消耗的时间并不确定，跟系统中还有多少熵尚未使用有关。如果没有熵可用，这个调用看上去就挂起了，可能一次长达数秒，直到有可用的熵为止。所以对性能的计时非常困难，因为性能本身也是随机的。

对于会创建很多 SSL 连接，或者需要大量安全随机数的应用而言，这往往会成为问题；这样的应用要花很多时间去执行其操作。当在一个这样的应用上执行性能测试时，请注意，计

注 2：这里的源类似于伪随机数中的初始种子。——译者注

时会有很多变数。除了像第 2 章讨论的那样运行大量示例测试，其实没什么办法处理此类变数。另一种选择是联系操作系统厂商，看他们是不是有更多（或更好的）基于熵的源。

必要时，可以使用第 3 种选择，即使用 `Random` 类运行性能测试，即便在生产环境中使用的是 `SecureRandom` 类。 如果性能测试是模块级的，这会很有意义：在同样的一段时间内，与产品系统相比，这些测试需要的随机数更多（比如需要更多 SSL 套接字）。但是，最终预期的负载必须用 `SecureRandom` 类来测试，以确定生产系统上的负载是否能够获得足够的随机数。

快速小结

1. Java 默认的 `Random` 类的初始化的成本很高，但是一旦初始化完毕，就可以重用。
2. 在多线程代码中，应该首选 `ThreadLocalRandom` 类。
3. `SecureRandom` 类表现出的性能也是随意的和完全随机的。在对用到这个类的代码做性能测试时，一定要认真规划。

12.4 Java原生接口

如果想编写尽可能快的代码，要避免使用 JNI。

在现行的 JVM 版本上，编写得好的 Java 代码至少会与相应的 C 或 C++ 代码跑得一样快（现在可不是 1996 年了）。语言纯粹主义者会继续争论 Java 和其他语言的相对性能指标，当然肯定能找到相应的例子，证明用其他语言编写的应用比用 Java 编写的相同应用快（不过这类例子中往往会包含写得很差的 Java 代码）。然而，这类争论并非本节的要领，这里要说的是：如果某个应用已经是用 Java 编写的，那出于性能原因调用原生代码几乎总是一个坏主意。

JNI 有时仍然非常有用。Java 平台提供了不同操作系统的很多公共特性，但如果需要访问一个特殊的、特定于操作系统的函数，那 JNI 就派上用场了。如果有现成的商用原生代码，那为什么还要构建自己的执行操作的库呢？在这种情况和其他一些情况下，问题就变成了如何编写最高效的 JNI 代码。

答案是尽可能避免从 Java 调用 C。跨 JNI 边界（边界是描述跨语言调用的术语）成本非常高，这是因为，调用一个现有的 C 库首先需要一些胶水代码，需要花时间通过胶水代码创建新的、粗粒度的接口，一下子要多次进入 C 库。

有趣的是，反过来就未必如此了：从 C 代码调用回 Java 不会有很大的性能损失（与所用的参数有关）。比如，考虑下面的代码：

```
public void main() {
    calculateError();
}

public void calculateError() {
    for (int i = 0; i < numberOfTrials; i++) {
       error += 50 - calc(numberOfIterations);
    }
```

```
    }

    public double calc(int n) {
        double sum = 0;
        for (int i = 0; i < n; i++) {
            int r = random(100); // 返回1至100之间的一个随机值
            sum += r;
        }
        return sum / n;
    }
```

这段（完全没有实际意义的）代码有两个主循环：内层循环多次调用生成随机数的代码，外层循环重复调用内层循环，看看所得的随机数与预期值（这里是 50）的接近程度。通过 JNI，可以用 C 实现 `calculateError()`、`calc()` 和 `random()` 这些方法中的任何一个或多个。表 12-4 展示了不同组合情况下的性能，其中 `numberOfTrials` 为 10 000。

表12-4：计算随机方法的error的时间

`calculateError`	`Calc`	`Random`	JNI转移	总时间（秒）
Java	Java	Java	0	12.4
Java	Java	C	10 000 000	32.1
Java	C	C	10 000	24.4
C	Java	Java	10 000	12.4
C	C	C	0	12.4

仅用 JNI 调用实现最内层方法，跨 JNI 边界的次数最多（`numberOfTrials * numberOfloops`，1 千万次）。将跨边界次数减少到 `numberOfTrials`（即 10 000）可以大幅减少开销，而将其减到 0，性能会最好——至少从 JNI 角度看是这样，尽管纯 Java 实现和完全使用原生代码一样快。

如果所用的参数不是简单的基本类型，JNI 代码性能会更糟。这一开销涉及两个方面。第一，对于简单的引用，需要地址转换。这也是为什么在上面的例子中，从 Java 调用 C 比从 C 调用 Java 开销更大：从 Java 调用 C，会隐式地把问题中的对象（`this`）传递给 C 函数，从 C 调用 Java 则无需传递任何对象。

第二，对于基于数组的数据，其中的操作在原生代码中会进行特殊处理。这包括 `String` 对象，因为字符串数据本质上是一个字符数组。要访问这类数组中的单个元素，必须调用一个特殊的方法，将该对象固定在内存中（对于 `String` 对象，要将其从 Java 的 UTF-16 编码转换成 UTF-8）。当不再需要数组时，必须在 JNI 代码中显式地释放。当有数组被固定在内存中时，垃圾收集器就无法运行——所以 JNI 代码中代价最高的错误之一就是在长期运行的代码中固定了一个字符串或数组。这会阻碍垃圾收集器运行，实际上也阻塞了所有应用线程，直到 JNI 代码完成。对于会固定数组的临界区，尽可能缩短固定时间极为重要。

有时，后面这个目标会与减少跨 JNI 边界调用这个目标冲突[3]。这种情况下，后一个目标更

注 3：这里是指，本来一次 JNI 调用可以完成的事情，因为要缩短固定数组的时间，所以可能要分成几次，以防影响垃圾收集器工作。——译者注

重要：即使这意味着要多次跨 JNI 边界，也要让固定数组和字符串的代码区尽可能短。

快速小结

1. JNI 并不能解决性能问题。Java 代码几乎总是比调用原生代码跑得快。
2. 当使用 JNI 时，应该限制从 Java 到 C 的调用次数；跨 JNI 边界的调用成本很高。
3. 使用数组或字符串的 JNI 代码必须固定这些对象；为避免影响垃圾收集器，应该限制固定对象的时间。

12.5 异常

Java 的异常处理一直有代价高昂的坏名声。其代价确实比处理正常的控制流高一些，不过在大多数情况下，这种代价并不值得浪费精力去绕过。另一方面，因为异常处理是有成本的，所以不应将其用作一种通用机制。这里的指导方针是，根据良好程序设计的一般原则来使用异常：基本上，代码仅应该通过抛出异常来说明发生了意料之外的情况。遵循良好的代码设计原则，意味着 Java 代码不会因异常处理而变慢。

有两个因素会影响异常处理的一般性能。一个是代码块本身：创建一个 try-catch 块代价高吗？尽管很久以前可能是这样，但是近几年来，情况已非如此。不过在互联网上有些信息会留存很久，所以偶尔还会看到有人建议避免使用异常，因为 try-catch 块代价较高。这些建议都是老黄历了，因为现代 JVM 生成的代码可以非常高效地处理异常。

第二个方面是，（大部分）异常会涉及获取该异常发生时的栈轨迹信息。这一操作代价可能会很高，特别是在栈的轨迹很深时。

下面看一个例子。假如现在有一个特定方法的 3 种实现：

```
public ArrayList<String> testSystemException() {
    ArrayList<String> al = new ArrayList<>();
    for (int i = 0; i < numTestLoops; i++) {
        Object o = null;
        if ((i % exceptionFactor) != 0) {
            o = new Object();
        }
        try {
            al.add(o.toString());
        } catch (NullPointerException npe) {
            // 继续获取下一个字符串
        }
    }
    return al;
}

public ArrayList<String> testCodeException() {
    ArrayList<String> al = new ArrayList<>();
    for (int i = 0; i < numTestLoops; i++) {
        try {
            if ((i % exceptionFactor) == 0) {
                throw new NullPointerException("Force Exception");
```

```
            }
            Object o = new Object();
            al.add(o.toString());
        } catch (NullPointerException npe) {
            // 继续获取下一个字符串
        }
    }
    return al;
}

public ArrayList<String> testDefensiveProgramming() {
    ArrayList<String> al = new ArrayList<>();
    for (int i = 0; i < numTestLoops; i++) {
        Object o = null;
        if ((i % exceptionFactor) != 0) {
            o = new Object();
        }
        if (o != null) {
            al.add(o.toString());
        }
    }
    return al;
}
```

每个方法都返回一个字符串数组，其元素是从新创建的对象得到的。数组的大小会变化，跟抛出异常的次数有关。

表 12-5 列出了在最坏情况下（即 `exceptionFactor` 为 1，每次迭代都会生成异常，得到的结果是一个空列表）为 100 000 次迭代执行每个方法的时间。示例代码中，有的方法栈轨迹很浅（当调用这个方法时，栈上只有 3 个类），有的栈轨迹很深（当调用这个方法时，栈上有 100 个类）。

表12-5：100%产生异常时的处理时间

方法	浅时间（毫秒）	深时间（毫秒）
代码异常	381	10 673
系统异常	15	15
防御性编程	2	2

这里有 3 点有趣的差别。首先，在每次迭代显式地构建异常的代码中，栈较浅和栈较深两种情况下时间差别很大。构建栈轨迹需要时间，这个时间和栈的深度有关。

第二个有趣的差别在这两种情况之间：代码显式地创建异常，或者是当 JVM 解析到空指针时创建异常（见表中的前两行）。目前的情况是，在某一个时刻，编译器会优化掉系统生成的异常；JVM 开始重用同一个异常对象，而不是每次需要时创建一个新的。不管调用栈是什么样的，相关的代码每次执行时都会重用这个对象；而且这个异常实际上没有包含调用栈（也就是说，`printStackTrace()` 没有输出）。这种优化在完整的栈异常信息抛出很长一段时间之后才会出现，所以如果测试用例中没有包含足够长的热身周期，是不会看到这种效果的。

最后，在访问对象之前先判断一下是否为 `null`，这种防御性编程性能最好。在这个例子中，这一点并不意外，因为整个循环变成了空操作。所以对这个数字要持保留态度。

尽管这些实现存在一些差别，但是请注意，大部分情况下，所用的时间都很少，是毫秒级的。平均到 100 000 次调用，每次调用的执行时间几乎看不到什么差别（别忘了，这还是最坏的情况）。

如果异常使用得当，这些循环中的异常数目就会非常小。表 12-6 列出了执行 100 000 次循环时，产生 1000 次异常（1% 的几率）需要的时间。

表12-6：有1%的几率产生异常时的处理时间

方法	浅时间（毫秒）	深时间（毫秒）
代码异常	56	157
系统异常	51	52
防御性编程	50	50

现在 `toString()` 方法的处理时间成了计算的大头。在栈较深的情况下，创建异常仍然有性能损失，不过提前测试 `null` 值的收益都被抵消了。

所以异常使用不当所带来的性能损失并没有想象的那么大。有些情况下，我们仍然会遇到创建太多异常的代码。因为性能损失主要来自填充栈轨迹信息，因此可以使用 `-XX:-StackTraceInThrowable` 标志（默认为 `false`）来禁止生成栈轨迹信息。

这并不是个好主意：栈轨迹的存在就是为帮我们分析哪里出问题的。如果使用了 `-XX:-StackTraceInThrowable` 标志，也就丢失了这种能力。而且有些代码实际上会检查栈轨迹，并以此确定如何从异常恢复。（CORBA 的参考实现就是这么工作的。）这种方式本身就有问题，但关键还在于禁止栈跟踪信息会使代码出现莫名其妙的问题。

JDK 中有些 API 的异常处理会导致性能问题。当集合中并不存在要检索的元素时，很多集合类就会抛出异常。比如 `Stack` 类，如果栈是空的，当调用 `pop()` 时，就会抛出 `EmptyStackException`。这种情况下，先通过防御性编程方式检查一下栈的长度会好一些。（另一方面，和很多集合类不同的是，`Stack` 类支持保存为 `null` 的对象，所以不能用 `pop()` 方法返回 `null` 来说明栈是空的。）

关于异常的不当使用，JDK 中最臭名昭著的例子是类加载：当使用 `ClassLoader` 类的 `loadClass()` 方法加载某个找不到的类时，就会抛出 `ClassNotFoundException`。这实际并不是一个异常条件。不要期望一个类加载器能知道如何加载应用中的每个类，这也是之所以会有类加载器的层次结构的原因了。

在一个存在大量类加载器的环境中，这意味着，在层次化的类加载器中搜索知道如何加载给定类的那个类加载器时，会有大量的异常。比如在本章前面类加载的例子中，如果关闭栈轨迹信息，运行速度会提升 3%。

不过，类加载只是个例外。那个例子是使用很长的 classpath 做的微基准测试，而且即便是在这样的条件下，每次调用的差别也是毫秒级的。

快速小结

1. 处理异常的代价未必会很高，不过还是应该在适合的时候才用。
2. 栈越深，处理异常的代价越高。
3. 对于会频繁创建的系统异常，JVM 会将栈上的性能损失优化掉。
4. 关闭异常中的栈轨迹信息，有时可以提高性能，不过这个过程往往会丢失一些关键信息。

12.6 字符串的性能

字符串对 Java 非常重要，其性能在其他章节也已经讨论过，这里再强调几点。

字符串保留

创建多个包含相同字符序列的字符串对象，这种情况很常见。没有必要在堆中为所有这些对象都分配空间；因为字符串是不可变的，所以重用现有的字符串往往更好。更多细节可参见第 7 章。

字符串编码

Java 的字符串采用的是 UTF-16 编码，而其他地方多是使用其他编码，所以将字符串编码到不同的字符集的操作很常见。对于 `Charset` 类的 `encode()` 和 `decode()` 方法而言，如果一次只处理一个或几个字符，它们会非常慢；务必完整缓存一些数据，再进行处理，本章前面也讨论过。

网络编码

在编码静态字符串（来自 JSP 文件等地方）时，Java EE 应用服务器往往会特殊处理；更多细节可参见第 10 章。

字符串连接是另一个可能会出现性能问题的地方。考虑这样一个简单的字符串连接操作：

```
String answer = integerPart + "." + mantissa;
```

这行代码实际上非常高效；`javac` 编译器的语法糖会将其转换为如下代码：

```
String answer = new StringBuilder(integerPart).append(".").
                                append(mantissa).toString();
```

不过问题来了，如果这个字符串是逐步构造起来的：

```
String answer = integerPart;
answer +=".";
answer += mantissa;
```

那么这段代码就会被翻译为：

```
String answer = new StringBuilder(integerPart).toString();
answer = new StringBuilder(answer).append(".").toString();
answer = new StringBuilder(answer).append(mantissa).toString();
```

所有那些临时的 `StringBuilder` 对象和中间的 `String` 对象都很低效。永远不要使用连接来构造字符串，除非能在逻辑意义上的一行代码内完成；也不要在循环内使用字符串

连接，除非连接后的字符串不会用于下一次循环迭代。对于其他情况，应该总是使用 `StringBuilder`，以获得更好的性能。在第 1 章中我曾经说过，某种情况下“过早的优化”只是表示要“编写良好的代码”。这就是最好的例子。

快速小结

1. 一行的字符串连接代码性能很不错。
2. 对于多行的连接操作，一定要确保使用 `StringBuilder`。

12.7 日志

日志是让程序员爱恨交织的事情之一。每当有人问为什么某个程序运行这么糟时，我首先要做的就是拿到任何可用的日志，希望在应用生成的日志中找到线索，以了解应用的状况。而每当有人让我审查工作代码的性能时，我又会立即建议把所有的日志语句关掉。

日志有很多种。GC 会生成自己的日志语句（参见第 6 章）。日志可以定向到一个单独的文件中，其大小可以由 JVM 管理。即便在生产代码中，GC 日志（使用 `-XX:+PrintGCDetails` 标志开启）的开销也是非常低的，而当出现问题时，它们的好处非常大，所以 GC 日志应该一直打开。

Java EE 应用服务器会生成一个访问日志，每当有请求时都会更新。这类日志的影响通常比较明显：不管在应用服务器上运行的是何种测试，关闭这类日志可以明显改进性能。根据我的经验，从诊断角度看，当出现问题时这些日志的帮助不是很大。不过在业务需求方面，这类日志往往非常关键，此时必须开启。

很多应用服务器都支持 Apache 的 `mod_log_config` 标准，尽管它并非一个 Java EE 标准。它可以针对每个请求精确地指定想要记录的信息（不支持 `mod_log_config` 语法的服务器通常也会支持某种形式的定制）。这里的关键是，记录的信息应该尽可能少，同时仍要满足业务需求。日志的性能会受所写数据量的影响。

特别是在 HTTP 访问日志中（或者笼统地说，在任何种类的日志中），记录下所有的数字信息是个不错的主意：记录 IP 地址而不是主机名，记录时间戳（比如从 Unix 纪元到现在所经过的秒数）而不是字符串数据（比如“Monday, June 3, 2013 17:23:00 -0600”），诸如此类。尽量减少需要花时间和内存去计算的任何数据转换，以便使日志对系统的影响将至最低。转换后的数据总是可以通过对日志做后续处理来获得。

对于应用日志，需要记住 3 个基本原则。第一，协调好要打日志的数据和所选级别（Level）之间的关系。JDK 中有 7 个标准的日志级别，而且 `Logger` 实例一般默认配置为输出其中的 3 个级别（`INFO` 及更高级别）。在项目中，这往往会导致混淆：`INFO` 级别听上去好像应该非常常见，而且应该提供与应用流程相关的描述（“现在正在处理 A 任务”，“现在正在做 B 任务”，等等）。特别是对于存在大量线程的可扩展应用（包括 Java EE 应用服务器）而言，这类日志多了会给性能带来不利影响（更不用说太多没什么用的日志信息带来的风险了）。要学会使用更低级别的日志语句。

类似地，当把代码签入到组库中时，应该考虑的是项目使用者的需求，而不是我们作为开

发者的需求。在将自己的代码集成到一个更大的系统中之后，这些代码运行状况如何呢？我们都希望获得与此相关的大量积极反馈，另外还会运行一连串的测试。但如果消息对最终用户或系统管理员没什么意义，那默认开启这些日志就没什么帮助。它们的“作用”不过是拖慢了系统（还会让最终用户迷惑不解）。

第二个原则是使用细粒度的 `Logger` 实例。对每个类的 `Logger` 实例进行配置可能会很繁琐，但这么做是值得的，因为能够更好地控制日志输出。在一个较小的模块中，让一组类共享一个 `Logger` 实例，是个不错的折中办法。要记住的关键一点是，如果生产环境变化很大，有些问题（特别是那些在高负载情况下出现的问题，或者是其他与性能有关的问题）很难重现。打开太多日志往往会改变环境，导致原来的问题不再复现。

因此，我们必须能够做到仅打开一小组代码的日志（至少最初能控制一小组 `FINE` 级别的日志语句，然后是控制更多 `FINER` 和 `FINEST` 级别的），这样就不会影响代码的性能了。

在这两个原则之间，应该能够支持在生产环境中生成信息的小子集，前提是不影响系统性能。无论如何这都是应该考虑的，原因在于：如果日志会让生产系统变慢，其管理员很可能不会开启日志；在这种情况下，如果系统确实变慢了，重现问题的可能性也小了。

第三个原则是，在向代码引入日志时，应该注意，很容易编写出带来意想不到的副作用的日志代码，即使这个日志并没有开启。这是可以说明“过早的优化”很不错的又一种情况：如第 1 章的例子所示，每当要打日志的信息包含方法调用、字符串连接或者其他任何形式的资源分配（比如为 `MessageFormat` 参数分配一个 `Object` 数组）时，记得使用 `isLoggable()` 方法。

快速小结

1. 为帮助用户找出问题，代码应该包含大量日志，但是这些日志默认都应该是关闭的。
2. 如果 `Logger` 实例的参数需要调用方法或者分配对象，那么在调用该实例之前，不要忘了测试日志级别。

12.8 Java集合类API

Java 的集合类 API 有很大的选择余地；Java 7 至少提供了 58 个不同的集合类。在编写应用时，选择恰当的集合类，以及恰当地使用集合类，是一个重要的性能考量。

使用集合类的第一条规则是，选择适合应用的算法需求的集合类。该建议并不是特定于 Java 的；这实际上是数据结构入门课程就会介绍的。`LinkedList` 不适合做搜索；如果需要访问一段随机的数据，应该将集合保存到 `HashMap` 中。如果数据需要有序排列，则应使用 `TreeMap`，而不是尝试在应用中做排序。如果会用索引访问数据，则使用 `ArrayList`；但如果会频繁地向该数组中间插入数据，则不要使用它，诸如此类。根据算法选择要使用哪个集合类，这非常重要，但是在 Java 中做选择和在其他编程语言中做选择并没有多少区别。

然而在使用 Java 的集合类时，还有一些特殊的地方需要考虑。

12.8.1 同步还是非同步

默认情况下，几乎所有的 Java 集合类都是非同步的（主要的例外是 `Hashtable`、`Vector` 及与其相关的类）。

> **同步的集合类**
>
> 如果想了解为什么 `Vector` 和 `Hashtable`（及相关类）是同步的，就得先来看一点历史。
>
> 在 Java 早期，它们是 JDK 中仅有的集合类。当时（在 Java 1.2 之前）还没有集合类框架（Collections Framework）的正式定义；它们只是最初的 Java 平台提供的几个有用的类。
>
> 在 Java 发布第一个版本时，大部分开发者对多线程知之甚少，而 Java 试图让开发者能够更容易地避免在多线程环境中编程的某些陷阱。因此，这些类就设计成了线程安全的。
>
> 遗憾的是，在早期的 Java 版本中，同步——甚至是不存在竞争时的同步——是个很大的性能问题，所以当第一个重大修订版本发布时，集合类框架采用了相反的做法：所有新的集合类默认都是非同步的。即使从那时开始同步性能已经有了显著提高，但仍然不是没有成本的；能够选择非同步的集合类，可以帮助大家编写更快的程序（偶尔会出现因并发修改某个非同步的集合而导致的 bug）。

第 9 章有一个微基准测试，比较了基于 CAS 的保护和传统的同步。这个例子在多线程的情况下不太实用，但如果问题中的数据只会由一个线程访问，又会怎么样呢？如果不使用任何同步，效果又会如何？表 12-7 列出了比较情况。因为这里没有试图考虑竞争，所以在这样一种情况下，这里的微基准测试是有效的：没有竞争，手头上的问题是研究同步访问资源有些多余时的损失。

表12-7：同步访问和非同步访问的性能

模式	总时间（秒）	平均到每次操作的时间（纳秒）
CAS 操作	6.6	13
同步方法	11.8	23
非同步方法	3.9	7.8

从第 2 列可以很明显地看出，与简单的非同步访问相比，如果使用了任何一种数据保护技术，都会有比较小的性能损失。然而，别忘了这是执行了 5 亿次操作的一个微基准测试，所以平均到每次操作，差别就只在 15 纳秒的量级上。如果相关操作在目标应用中执行得足够频繁，性能损失就会有点明显。在大部分情况下，这种差别会被应用中其他更为低效的地方抵消掉。还要记住，这里的绝对数字完全是由测试所运行的目标机器决定的（我的是搭载了 4 核 AMD Athlon 处理器的家用机）；要获得更为真实的测量结果，测试需要在与目标环境相同的硬件上运行。

那么，如果要在同步的 `Vector` 和非同步的 `ArrayList` 之间做出选择，该选择哪个呢？访问 `ArrayList` 会稍微快一些，这与访问这个列表的频繁程度有关，性能差异是可以测量的。

（正如第 9 章所指出的，在某些硬件平台上，过多调用同步方法对性能影响很大。）

另一方面，这里假设代码不会被多个线程访问。今天可能确实如此，那明天会怎么样呢？如果情况可能会变，那更好的办法是现在就使用同步的集合，并减轻它所带来的性能影响。这是一个设计选择，为使代码经受住时间的考验而将其设计为线程安全的，在这上面投入时间和精力是不是值得，取决于开发应用时的情况。

如果要在非同步集合和使用了 CAS 法则的集合之间做出选择（比如在 `HashMap` 和 `ConcurrentHashMap` 之间），它们的性能差别会微乎其微。当基于 CAS 的类用于不存在竞争的环境中时，几乎没有什么开销（继续阅读，后面会讨论特定的 hashmap 在内存使用方面的差别）。

12.8.2 设定集合的大小

集合类的用途是保存任意数量的数据元素，并随着集合中新条目的添加，在必要时进行扩展。性能方面有两点需要考虑。

尽管 Java 中的集合类提供的数据类型非常丰富，但是在基本层面上，这些类都必须仅使用 Java 基本的数据类型来保存其数据：数值（整型、双精度浮点型等）、对象引用和这些类型的数组。因此，`ArrayList` 中包含一个真正的数组：

```
private transient Object[] elementData;
```

随着在 `ArrayList` 中添加和移除条目，这些条目会保存在 `elementData` 数组内的期望位置（可能会导致数组中的其他条目变更位置）。类似地，`HashMap` 中包含着一个由内部数据类型 `HashMap$Entry` 组成的数组，`HashMap` 会将每个键 - 值对映射到这个数组中，具体位置根据键的哈希码值来确定。

并非所有的集合类都使用数组保存其元素；比如 `LinkedList`，它以内部定义的 `Node` 类保存每个数据元素。但是使用数组保存元素的集合类都会涉及一个问题，就是要考虑数组的大小。如何确定某个特定的类是不是属于这个范畴呢？可以看看它的构造函数：如果它有一个构造函数支持指定该集合的初始大小，那它内部就使用了某个数组来存储元素。

对于这样的集合类，精确地指定初始大小非常重要。以 `ArrayList` 作为一个简单的例子：`elementData` 数组默认的初始大小为 10。当向某个 `ArrayList` 实例中插入第 11 个元素时，它就会扩展 `elementData` 数组。这意味着分配一个新数组，将原来的内容复制到这个数组中，然后添加新元素。可以说 `HashMap` 使用的数据结构和算法更复杂一些，但基本原理是一样的：在某一时刻，必须重新调整内部数据结构的大小。

`ArrayList` 类调整数组大小的方法是，在现有基础上增加约一半。所以 `elementData` 数组的大小最初是 10，然后是 15，22，33，以此类推。不管使用何种算法调整数组大小（参见后面方框内的文字），都会导致一些内存被浪费（这反过来又会影响应用花在执行 GC 上的时间）。此外，每当数组必须调整大小时，都伴随一个成本很高的数组复制操作，将老数组中的内容转移到新数组中。

要减少这些性能损失，必须尽可能准确地估计一下集合最终的大小，并用这个值来构建集合。

> **非集合类中的数据扩展**
>
> 很多非集合类也会在内部数组中保存大量数据。比如，`ByteArrayOutputStream` 类必须把写入到该流中的所有数据保存到一个内部缓冲区中；类似地，`StringBuilder` 和 `StringBuffer` 类也必须将所有字符保存到一个内部的字符数组中。
>
> 这些类大多会使用同样的算法调整内部数组的大小：需要调整时就加倍。这意味着，平均而言，内部的数组要比当前包含的数据多 25%。
>
> 这里的性能考量是相似的：使用的内存量多于 `ArrayList` 这个例子，需要复制数据的次数要少一些，但原理是一样的。在构建某个对象时，如果可以设置其大小，可以评估一下这个对象最终会保存多少数据，然后选择接受大小参数的那个构造函数。

12.8.3 集合与内存使用效率

我们刚看了一个集合的内存使用效率没有达到最佳的例子：在用于保存集合中的元素的底层存储中，往往会浪费一些内存。

对于元素比较稀疏的集合（只有一两个元素），这存在较大的问题。如果这样的集合用得非常多，则会浪费大量内存。解决方案之一就是在创建集合时设定其大小。另一种方案是，考虑一下这种情况是不是真的需要集合。

大部分开发者被问及如何快速地排序任意一个数组时，答案都会是快速排序（quicksort）。而好的性能工程师希望了解数组的大小：如果数组足够小，那最快的方式是使用插入排序（insertion sort）。（对于较小的数组来说，基于快速排序的算法通常会使用插入排序；就 Java 而言，`Arrays.sort()` 方法的实现就假定，少于 47 个元素的数组用插入排序比用快速排序更快。）数组大小至关重要。

> **JDK 7u40 中集合类内存大小**
>
> 很多应用中经常出现因集合类使用不当而导致的问题，所以 JDK 7u40 向 `ArrayList` 和 `HashMap` 的实现中引入了一个新的优化：默认情况下（比如在调用构造函数时没有使用大小参数），这些类不再为数据分配任何底层存储，而是在向该集合中插入第一个元素时才分配。
>
> 这就是第 7 章所探讨的延迟初始化技术的一个例子，在测试一些常见的应用时，因为减少了对 GC 的需求，所以性能有所改进。这些应用中有很多从来没用过的集合；所以延迟分配其底层存储在性能方面有优势。因为每次访问时本来就要检查底层存储的大小，所以检查底层存储是不是已经分配，并没有性能损失（不过创建最初的底层存储所需要的时间从创建对象时变成了向对象中插入第一个数据时）。

类似地，在基于某个键值查找数据时，`HashMap` 是最快的；但如果只有一个键，与使用一个简单的对象引用相比，使用 `HashMap` 就是大材小用了。即使有几个键，维护几个对象引用所需要的内存也比一个完整的 `HashMap` 对象少，而且这样对 GC 也有积极的影响。

除了以上这些，关于集合类的内存使用还有很重要的一点区别需要了解，那就是 HashMap 对象和 ConcurrentHashMap 对象大小的差别。在 Java 7 之前，一个空的或者元素稀疏的 ConcurrentHashMap 对象非常大：超过 1 KB（即便向其构造函数传了一个很小的大小）。在 Java 7 中，其大小只有 208 字节（与之相比，构造时没有指定大小的空 HashMap 占 128 字节，指定大小为 1 的 HashMap 占 72 字节）。

在存在很多小型 Map 的应用中，大小的差别仍然非常重要，但是 Java 7 中引入的优化使得这种差别不那么显著了。为提高性能，在内存非常重要且存在大量 Map 的应用中，有人（包括我）建议避免使用 ConcurrentHashMap 类。这些建议的核心其实是两个因素之间的取舍：是要更快地访问 Map（如果存在竞争），还是要小心更大的 Map 所引发的对垃圾收集器的压力。这个取舍如今仍然存在，但重心已更多地偏向了使用 ConcurrentHashMap。

快速小结

1. 仔细考虑如何访问集合，并为其选择恰当的同步类型。不过，在不存在竞争的条件下访问使用了内存保护的集合（特别是使用了基于 CAS 的保护的集合），性能损失会极小；有时候，保证安全性才是上策。
2. 设定集合的大小对性能影响很大：集合太大，会使得垃圾收集器变慢；集合太小，又会导致大量的大小调整与复制。

12.9 AggressiveOpts标志

AggressiveOpts 标志（默认为 false）会影响一些基本 Java 操作的行为。其目标是试验性地引入一些优化；随着时间的推移，原来由这个标志启用的优化有望成为 JVM 的默认设置。很多 Java 6 中的这类优化在 Java 7u4 中都成了默认的设置。在每个 JDK 版本中，都应该重新测试该标志，看看它对应用是否还有积极的影响。

12.9.1 替代实现

启用 AggressiveOpts 标志的主要影响是，它会为 JDK 中的一些基本的类引入不同的替代实现：尤其是 java.math 包中的 BigDecimal、BigInteger 和 MutableBigDecimal 类；java.text 中的 DecimalFormat、DigitalList 和 NumberFormat 类；java.util 包中的 HashMap、LinkedHashMap 和 TreeMap 类。

这些类在功能上与它们所替代的标准 JDK 中的类是一致的，但是采用了更高效的实现。在 Java 8 中，这些替代实现已经去掉，或者是合并到了基本的 JDK 类中，或者是以其他方式改进了基本的 JDK 类。

之所以要通过设置 AggressiveOpts 标志才能启用这些类（在 Java 7 中），原因在于它们的行为可能会在应用代码中引发一些微妙的 bug。比如，HashMap 类的激进实现所生成的迭代器，其返回键值的顺序与标准实现相比会有所不同。应用首先绝不能依赖该迭代器返回元素的顺序，但现实中很多应用都会犯这个错误。出于兼容性考虑，这种更高效的实现并没有覆盖掉原来的实现，所以必须通过设置 AggressiveOpts 标志来获得更好的性能。

因为 Java 8 去掉了这些类，所以在升级时可能会损害到 bug 兼容性——这又一次暗示了从一开始就写好代码的重要性。

12.9.2　其他标志

开启 `AggressiveOpts` 标志会影响其他一些较为次要的 JVM 调优。

设置 `AggressiveOpts` 标志会开启 `Autofill` 标志（它在 JDK 7 到 7u4 这几个版本中默认为 `false`）。这个标志开启后，编译器会对循环进行更好的优化。类似地，`AggressiveOpts` 标志还会开启 `DoEscapeAnalysis` 标志（在 JDK 7u4 及后续版本中，这个标志也成了默认的）。

`AutoBoxCacheMax` 标志（默认为 128）被设置为 20 000，支持对更多值进行自动装箱，这会轻微改进特定应用的性能（代价是使用的内存也会稍微增多）。`BiasedLockingStartupDelay` 会从默认的 2000 减到 500，这意味着偏向锁会在应用开始执行后更短的时间内启用。

最后，该标志还会开启 `OptimizeStringConcat` 标志，允许 JVM 优化 `StringBuilder` 对象的使用，具体而言，当编写下面这样的代码时，`javac` 编译器会创建 `StringBuilder` 对象：

```
String s = obj1 + ":" + obj2 + ":" + obj3;
```

`javac` 编译器会将这行代码翻译为一个 `StringBuilder` 对象上的一系列 `append()` 调用。当 `OptimizeStringConcat` 标志开启时，JVM 即时编译器（JIT）会把 `StringBuilder` 对象的创建优化掉。在 JDK 7 到 7u4 的版本中，`OptimizeStringConcat` 标志默认为 `false`，而在开启 `AggressiveOpts` 标志后，其默认值变成了 `true`。

快速小结

1. `AggressiveOpts` 标志会在一些基本的类中开启某些优化。大多数情况下，这些类要快于它们所替代的类，不过可能有意想不到的副作用。
2. Java 8 中去掉了这些类。

12.10　Lambda表达式和匿名类

对很多开发者而言，Java 8 最激动人心的特性就是加入了 Lambda 表达式。不可否认，Lambda 对 Java 开发者的开发效率有着非常积极的影响，尽管收益难以量化，但是我们可以使用 Lambda 表达式来考查代码的性能。

关于 Lambda 表达式的性能，一个最基本的问题是，它们与其所对应的替代物匿名类相比如何。其实几乎没什么差别。关于如何使用 Lambda 表达式，常见的例子一般是从创建匿名内部类的代码入手（不过这类例子往往使用 `Stream`，而不是像下面这样使用迭代器；12.11 节会介绍 `Stream` 类）：

```
private volatile int sum;

public interface IntegerInterface {
    int getInt();
}
```

```
public void calc() {
    IntegerInterface a1 = new IntegerInterface() {
        public int getInt() {
            return 1;
        }
    };
    IntegerInterface a2 = new IntegerInterface() {
        public int getInt() {
            return 2;
        }
    };
    IntegerInterface a3 = new IntegerInterface() {
        public int getInt() {
            return 3;
        }
    };
    sum = a1.get() + a2.get() + a3.get();
}
```

可以将其与下面使用了 Lambda 表达式的代码对比一下：

```
public void calc() {
    IntegerInterface a3 -> { return 3 };
    IntegerInterface a2 -> { return 2 };
    IntegerInterface a1 -> { return 1 };
    sum = a3.get() + a2.get() + a1.get();
}
```

这里 Lambda 表达式或匿名类的代码体至关重要：如果其中执行了任何较为重型的操作，那花在这一操作上的时间会把 Lambda 表达式或匿名类实现上的细微差距掩盖掉。然而，即便在这种最简单的情况下，执行该操作的时间也基本一样，如表 12-8 所示。

表12-8：使用Lambda表达式和匿名类执行`calc()`方法的时间

实现方式	所用时间（微秒）
匿名类	87.2
Lambda 表达式	87.9

数字看上去比较正式，让人印象深刻，但除了说这两种实现性能基本相同，我们也得不出其他结论。确实如此，因为测试中存在随机波动，再加上这些调用都是用 `System.nanoTime()` 测量的。在这个层次上，这样计时还没有准确到足以让人信服；总而言之，我们所知道的就是它们的性能相同。

在这个例子中的典型用法中，有一点比较有趣，即每当方法被调用时，使用匿名类的代码都会创建一个新对象。如果这个方法调用次数非常多（当然必须在某个基准测试中测量其性能），会有很多这个匿名类的对象被快速创建并丢弃。如第 5 章所介绍，这种用法对性能几乎没有什么影响。分配对象（以及更重要的初始化操作）的成本非常低，而且因为它们很快就会被丢弃，实际上不会拖慢垃圾收集器。

尽管如此，我们总是可以构造一些用例，来说明分配对性能影响很大，以及最好重用对象：

```
private IntegerInterface a1 = new IntegerInterface() {
    public int getInt() {
        return 1;
    }
};
…… 其他接口类似……
public void calc() {
        return a1.get() + a2.get() + a3.get();
    }
}
```

而 Lambda 表达式的这种典型用法，通常不会在每次循环迭代时创建一个新对象，所以在个别案例下，使用 Lambda 表达式的性能会好一些。尽管如此，即便要构造性能差异有影响的微基准测试，都是非常困难的。

Lambda表达式与匿名类加载

有种极端情况，即在启动和类加载时，两种实现的性能差别很明显。人们很容易查看 Lambda 表达式的代码，并断定它不过是语法糖，底层还是创建匿名类（特别是从长远来看，两者的性能一样）。但现在的工作方式并不是这样的。在 JDK 8 中，Lambda 表达式的代码会创建一个静态方法，这个方法通过一个特殊的辅助类来调用。而匿名类是一个真正的 Java 类，有单独的 class 文件，并通过类加载器加载。

如本章前面所介绍的，类加载的性能可能很重要，特别是在 classpath 很长的情况下。如果这个例子就是在这样的情况下运行——`calc()` 方法每次都在一个新的类加载器中执行，那匿名类实现就处于劣势了。表 12-9 列出了这种情况下的差别。

表12-9：在一个新的类加载器中执行`calc()`方法的时间

实现方式	所用时间（微秒）
匿名类	267
Lambda 表达式	181

关于这些数字，有一点要提一下：它们都是在经过一段适当的热身周期（以开启编译）之后再测量的。但是在热身阶段会发生另一件事：class 文件第一次被从磁盘读取出来。操作系统会把这些文件保存在内存（操作系统的文件缓冲区）中。所以代码第一次执行需要的时间比较长，因为要通过读文件的系统调用把文件从磁盘中真正地加载进来。随后的调用会快很多：尽管仍然需要通过系统调用读文件，但因为这些文件已经在操作系统的内存中，所以数据可以快速返回。因此，匿名类实现的性能可能要比想象中好，因为它并没有真正地从磁盘读取 class 文件。

快速小结

1. 如果要在 Lambda 表达式和匿名类之间做出选择，则应该从方便编程的角度出发，因为性能上没什么差别。
2. Lambda 表达式并没有实现为类，所以有个例外情况，即当类加载行为对性能影响很大时，Lambda 表达式略胜一筹。

12.11 流和过滤器的性能

新的 Stream 设施是 Java 8 的另一个关键特性，而且经常与 Lambda 表达式配合使用。在性能方面，流有一个很重要的特性，即它们可以自动并行化代码。关于并行流的信息可参见第 9 章；本节将探讨流和过滤器的一般性能特性。

延迟遍历（Lazy Traversal）

Stream 的第一个性能优势是它们被实现为了延迟的数据结构。举个例子，有一组股票代码，我们想从其中找到第一个不含字母 A 的代码。通过流实现该功能的代码看上去就像下面这样：

```
public String findSymbol(ArrayList<String> al) {
    Optional<String> t = al.stream().
            filter(symbol -> symbol.charAt(0) != 'A').
            filter(symbol -> symbol.charAt(1) != 'A').
            filter(symbol -> symbol.charAt(2) != 'A').
            filter(symbol -> symbol.charAt(3) != 'A').
               findFirst();
    return t.get();
}
```

很明显，用一个过滤器实现会更好，不过我们还是稍后讨论。现在先思考一下，对例子中的这个流而言，实现为延迟数据结构有何意义？每个 filter() 方法都会返回一个新流，所以这里实际上有 4 个逻辑流。

其实除了设置一系列指针，filter() 方法什么都没做。其作用是，当在这个流上调用 findfirst() 方法时，不会执行任何的数据处理，也不会拿数据去跟字符 A 作比较。

相反，findfirst() 会向前面的流（从第 4 个过滤器返回的那个）要一个元素。而那个流还没有元素，于是又向后调用由第 3 个过滤器生成的流，以此类推。第 1 个过滤器将从 ArrayList（从技术上讲，就是从原始的流中）抓到第一个元素，并测试它的第一个字符是否不为 A。如果确实不是，它会完成回调，并将该股票代码向后返回；否则，它会继续迭代数组，直到找到一个匹配的股票代码（或者找遍整个数组）。第 2 个过滤器行为类似——当对第 1 个过滤器的回调返回时，它会测试第二个字符是否不为 A。如果确实不是，它会完成回调，并将该股票代码向后传递；否则，调用第 1 个过滤器获得下一个股票代码。

这么多回调听上去效率很低，那就考虑一个替代方案。尽早处理流的算法类似这样：

```
private <T> ArrayList<T> calcArray(ArrayLisr<T> src, Predicate<T> p) {
    ArrayList<T> dst = new ArrayList<>();
    for (T s : src) {
        if (p.test(s))
            dst.add(s);
    }
    return dst;
}
```

```
private static long calcEager(ArrayList<String> a1) {
    long then = System.currentTimeMillis();
    ArrayList<String> a2 = calcArray(a1, (String s) -> s.charAt(0) != 'A');
    ArrayList<String> a3 = calcArray(a2, (String s) -> s.charAt(1) != 'A');
    ArrayList<String> a4 = calcArray(a3, (String s) -> s.charAt(2) != 'A');
    ArrayList<String> a5 = calcArray(a4, (String s) -> s.charAt(3) != 'A');
    answer = a5.get(0);
    long now = System.currentTimeMillis();
    return now - then;
}
```

与 Java 实际采用的延迟实现相比，这一替代方案效率要差些，原因有二。第一，它需要创建大量临时的 `ArrayList` 实例。第二，在延迟实现中，一旦 `findfirst()` 方法得到一个元素，处理就可以停止了。这意味着实际通过过滤器传递的只是所有元素的一个子集。而另一方面，在尽早处理的实现方案中，则必须多次处理整个 `ArrayList`，一直到最后一个创建。

因此，在这个例子中延迟实现比上面的替代方案性能更好，也就不足为奇了。具体而言，测试中所处理的 `ArrayList` 包含 456 976 个元素，每个元素都是 4 字母股票代码，元素按字母顺序排列。就延迟实现而言，在遇到 BBBB 之前，只处理了 18 278 个元素，到 `BBBB` 就可以停止了。迭代器实现所花的时间要比延迟实现长两个数量级，如表 12-10 所示。

表12-10：延迟处理与尽早处理的时间对比

实现	所用时间（秒）
过滤器 /`findFirst`	0.359
迭代器 /`findFirst`	48.706

为什么过滤器解决方案比迭代器快这么多呢？一个原因是，过滤器有机会使用算法优化：当完成需要做的任务时，就可以停下来，因此处理的数据较少。

如果必须处理整组数据，过滤器和迭代器的性能相比又会如何呢？对于这个例子，我们稍微修改一下测试。前面的例子很好地演示了多个过滤器如何工作，但是很明显，如果用一个过滤器处理，性能有望变得更好：

```
public int countSymbols(ArrayList<String> al) {
    int count = 0;
    t = al.stream().
            filter(symbol -> symbol.charAt(0) != 'A' &&
                             symbol.charAt(1) != 'A' &&
                             symbol.charAt(2) != 'A' &&
                             symbol.charAt(3) != 'A').
               forEach(symbol -> count++);
    return count;
}
```

这个例子也修改了最终代码，会计算股票代码的个数，这样就会处理整个列表了。另一方面，尽早处理的方案也可以稍作修改，直接使用迭代器：

```
public int countSymbols(ArrayList<String> al) {
    int count = 0;
```

```
        for (String symbol : al) {
            if (symbol.charAt(0) != 'A' &&
                symbol.charAt(1) != 'A' &&
                symbol.charAt(2) != 'A' &&
                symbol.charAt(3) != 'A')
                count++;
        return count;
    }
```

即使在这种情况下，延迟过滤器的实现还是要比迭代器快（参见表 12-11）。

表12-11：使用一个过滤器与一个迭代器的时间

实现	所用时间（秒）
多个过滤器	18.0
单个过滤器	6.5
迭代器 / 计数	6.8

出于比较的目的，表 12-11 的第一行是使用 4 个独立的迭代器处理整个列表的情况。和只用一个过滤器这种最优情况相比，使用一个过滤器还是要比使用一个迭代器稍微快些。

快速小结

1. 过滤器因为支持在迭代过程中结束处理，所以有很大的性能优势。
2. 即使都要处理整个数据集，一个过滤器还是要比一个迭代器稍微快些。
3. 多个过滤器有些开销，所以要确保编写好用的过滤器。

12.12 小结

本章研究了 Java SE JDK 的一些重点领域，并对 Java 性能方面的一些实验做了总结。本章中的大部分主题都有一个有趣之处，即它们表现出了 JDK 本身性能的演进。随着 Java 作为一个平台不断发展和成熟，Java 的开发者发现：对于会重复生成的异常，不需要浪费时间提供线程栈；使用线程局部变量避免随机数生成器的同步，是个不错的选择；`ConcurrentHashMap` 默认大小已经太过庞大；类加载器因为同步锁而不能并行运行；诸如此类。

成功实现改进的持续过程都与 Java 的性能优化息息相关：从调优编译器和垃圾收集器，到更高效地使用内存，再到理解 Java SE 和 Java EE API 的主要性能差异等。学习了本书中的各种工具和过程以后，你就可以对自己的代码运筹帷幄，实现持续的改进。

附录A

性能调优标志摘要

表A-1：调优JIT编译器的标志

标志	功能	使用场景	更多信息
-server	选择使用 server 编译器	适用于需长时间运行，同时又要求高性能的应用	“热点编译”，P59
-client	选择使用 client 编译器	适用于启动时间是最重要因素的应用	“热点编译”，P59
-XX:+TieredCompilation	使用 tiered 编译（同时选择 client 和 server 标志的特性）	适用于希望取得最佳性能，同时又有足够的内存可以支撑额外的编译代码的应用	“热点编译”，P59
-XX:ReservedCodeCacheSize=<*MB*>	设定 JIT 编译器进行代码编译的保留空间	当你遭遇警示消息“你的代码缓存已用尽”时，可以使用这个标志，通常结合 Tiered 编译标志一起使用	“调优代码缓存”，P67
-XX:InitialCodeCacheSize=<*MB*>	用于替 JIT 编译器编译代码分配初始空间	如果你需要为代码缓存预分配内存，可以使用该标志，不过这种情况不常发生	“调优代码缓存”，P67
-XX:CompileThreshold=<*N*>	设置一段代码或循环执行多少次之后转而进行编译	使用 server 编译器时，调整这个标志可以让更多的方法变为编译执行，让编译更早地发生。如果你使用的不是 Tiered 编译，第一种情况有时就是一种优势。	“编译阈值”，P68
-XX:+PrintCompilation	在日志中输出 JIT 编译器进行的操作	如果你怀疑某个重要的方法没有进行编译，或者对编译器的工作流程感到好奇，都可以使用这个标志进行查看	“检测编译过程”，P70

（续）

标志	功能	使用场景	更多信息
`-XX:+CICompilerCount=<N>`	设置 JIT 编译器使用的线程数	使用 Tiered 编译标志时，如果有太多的编译线程启动（尤其是在运行了多个 JVM 的大型机上），通过该标志可以调节 JIT 线程的数目	“编译线程”，P73

表A-2：选择GC算法的标志

标志	功能	使用场景	更多信息
`-XX:+UseSerialGC`	选择使用简单的单线程垃圾收集算法	适用于容量小于 100 MB 的堆	“Serial 垃圾收集器”，P84
`-XX:+UseParallelOldGC`	应用线程停顿时使用多个线程对老年代进行垃圾回收	如果你的应用能够容忍偶尔发生的长时间停顿，而你又希望用最小的 CPU 消耗取得最大的吞吐量，可以考虑使用这个标志	“Throughput 收集器”，P85
`-XX:+UseParallelGC`	应用线程停顿时使用多个线程对新生代空间进行垃圾收集	与 `UseParallelGC` 标志结合使用	“Throughput 收集器”，P85
`-XX:+UseConcMarkSweepGC`	使用后台线程以最低的停顿时间回收老年代的垃圾对象	如果你有充足的 CPU 资源可以运行后台线程，你的堆也相对较小，并且你不希望垃圾收集的停顿时间过长，就可以使用该标志	“CMS 收集器”，P85
`-XX:+UseParNewGC`	应用线程停顿时，使用多个线程回收新生代空间	与 `ConcMarkSweepGC` 标志结合使用	“CMS 收集器”，P85
`-XX:+UseG1GC`	应用线程停顿时，使用多个线程回收新生代空间，使用后台线程回收老年代空间，以最大程度减小停顿	你有足够的 CPU 资源可以运行后台线程，你的堆也比较大，同时你又不希望有长时间的停顿，可以考虑使用该标志	“G1 垃圾收集器”，P85

表A-3：适用于所有GC算法的通用标志

标志	功能	使用场景	更多信息
`-Xms`	设定堆的初始大小	如果堆的默认初始大小对于你的应用而言过小，可以考虑使用该标志	“调整堆的大小”，P92
`-Xmx`	设定堆的最大值	如果默认的堆大小对你的应用而言过小（或者过大），可以利用该标志调整	“调整堆的大小”，P92

（续）

标志	功能	使用场景	更多信息
-XX:NewRatio	设置新生代与老年代之间的比率	增大这个值会降低分配给新生代空间的堆的比例；减小这个值可以增大分配给新生代的堆的比率。这个设置只是一个初始值；除非自适应调整被关闭了，否则这个比率会随着垃圾收集发生变化（CMS 收集器是个例外，使用 CMS 时新生代的大小保持恒定）。随着新生代空间的减小，新生代垃圾收集的频率会增加，Full GC 的频率会降低（反之亦然）	“代空间的调整”，P95
-XX:NewSize	设置新生代的初始大小	你已经清楚地了解你应用程序的需求时，就可以进行相应的设置	“代空间的调整”，P95
-XX:MaxNewSize	设置新生代的最大值	你已经清楚地了解你应用程序的需求时，就可以进行相应的设置	“代空间的调整”，P95
-Xmn	设置新生代的初始值最大值	你已经清楚地了解你应用程序的需求时，就可以进行相应的设置	“代空间的调整”，P95
-XX:PermSize=*N*（只适用于 JDK 7）	设置永久代的初始值	如果你的应用使用了大量的类，你可能需要适当调整，增大默认值	“永久代和元空间的调整”，P96
-XX:MaxPermSize=*N*（只适用于 JDK 7）	设置永久代的最大值	如果你的应用使用了大量的类，你可能需要适当调整，增大默认值	“永久代和元空间的调整”，P96
-XX:MetaspaceSize=*N*（只适用于 JDK 8）	设置元空间的初始大小	如果你的应用使用了大量的类，你可能需要适当调整，增大默认值	“永久代和元空间的调整”，P96
-XX:MaxMetaspaceSize=*N*（只适用于 JDK 8）	设置元空间的最大容量	减小这个值可以限制类的元数据占用空间大小	“永久代和元空间的调整”，P96
-XX:ParallelGCThreads=*N*	设置垃圾收集器使用的线程数	如果系统上同时运行了多个 JVM，可以适当减小该参数值。如果 JVM 使用的堆非常大，又运行在一个处理能力很强的机器上，就应该适当增大这个参数值	“控制并发”，P97
-verbose:gc	开启基本的 GC 日志功能	GC 日志应一直开启，但是这还不够，通常更详细的日志会更好	“GC 工具”，P99

（续）

标志	功能	使用场景	更多信息
-Xloggc:<*path*>	将 GC 日志记录到某个文件而非标准输出	亘古不变的是，在日志中保存这些信息总是更好的	“GC 工具”，P99
-XX:+PrintGC	开启 GC 的基本日志	我们应该尽量开启 GC 日志，越详细的日志通常越有益	“GC 工具”，P99
-XX:+PrintGCDetails	开启 GC 的详细日志	即使是在生产环境中，也应尽量开启该标志（GC 日志的开销几乎可以忽略）	“GC 工具”，P99
-XX:+PrintGCTimeStamps	为 GC 日志中的每个条目打印相对时间戳	尽量开启，除非你已经开启了日期时间戳	“GC 工具”，P99
-XX:+PrintGCDateStamps	为 GC 日志中的每个条目打印日期时间戳	相对于时间戳方式，这种方式的开销更大，不过可能更容易处理	“GC 工具”，P99
-XX:+PrintReferenceGC	打印 GC 过程中弱引用和软引用的信息	如果程序中大量使用了这种引用，开启这个标志可以了解它们对 GC 开销的影响	“弱引用、软引用与其他引用”，P165
-XX:+UseGCLogFileRotation	开启 GC 日志循环功能以节省文件空间	在长期持续运行的生产环境中，GC 日志会消耗大量的磁盘空间。在这种情况下，你可能希望开启该标志	“GC 工具”，P99
-XX:NumberOfGCLogFiles=*N*	开启 GC 日志文件循环时，你可以使用该标志设定保留多少个日志文件	在长期持续运行的生产环境中，GC 日志会消耗大量的磁盘空间。在这种情况下，你可能希望开启该标志	“GC 工具”，P99
-XX:GCLogFileSize=*N*	开启日志循环时，利用该标志可以设定日志循环之前每个日志文件的大小	需要长期持续运行的生产环境中 GC 日志会消耗大量的磁盘空间，这种情况你可能希望开启该标志	“GC 工具”，P99
-XX:+UseAdaptiveSizePolicy	设置该标志时，JVM 会依据设定的 GC 目标调整堆的大小	如果堆的大小已经精细调优过，则可以关闭该标志	“自适应调整”，P98
-XX:+PrintAdaptiveSizePolicy	在 GC 日志中输出代大小调整的详细信息	通过该标志我们可以了解 JVM 是如何工作的。使用 G1 收集器时，通过该标志的输出可以了解 Full GC 是否源于巨型对象的分配	“自适应调整”，P98
-XX:+PrintTenuringDistribution	在 GC 日志中输出对象保持的信息	通过对象保持的信息我们可以判断是否需要调整对象保持的标志，以及如何调整这些标志	“晋升及 Survivor 空间”，P126
-XX:InitialSurvivorRatio=*N*	设置新生代中预留给 Survivor 空间的大小	如果“短寿”（short-lived）对象频繁地晋升到老年代，你可能需要考虑增大这个值	“晋升及 Survivor 空间”，P126

（续）

标志	功能	使用场景	更多信息
-XX:MinSurvivorRatio=N	设置新生代空间中用于自适应调整的 Survivor 空间大小	减小这个值会减小 Survivor 空间的最大值（反之亦然）	“晋升及 Survivor 空间”，P126
-XX:TargetSurvivorRatio=N	JVM 试图在 Survivor 空间保留的空闲空间	增大这个值会缩减 Survivor 空间的大小（反之亦然）	“晋升及 Survivor 空间”，P126
-XX:InitialTenuringThreshold=N	JVM 试图在 Survivor 空间保持对象的 GC 周期数，该参数设置的是一个初始值	增大这个值可以让对象在 Survivor 空间停留更长的时间，不过，需要注意的是，JVM 也会对该参数进行调节	“晋升及 Survivor 空间”，P126
-XX:MaxTenuringThreshold=N	设置 JVM 可以将一个对象保持在 Survivor 空间的最大 GC 周期数	增大这个值可以让对象在 Survivor 空间停留更长时间；JVM 会依据这个设定值与初始阈值，对实际的晋升阈值进行调整	“晋升及 Survivor 空间”，P126

表A-4：Throughput收集器调优标志

标志	功能	使用场景	更多信息
-XX:MaxGCPauseMillis=N	为 Throughput 收集器设定最长停顿时间，堆的大小会依据该目标动态调整	通常作为 Throughput 收集器调优的第一步，如果 Throughput 收集器计算出的默认堆大小无法达到应用的目标就会对其进行调整	“堆大小的自适应调整和静态调整”，P105
-XX:GCTimeRatio=N	用于控制 Throughput 收集器在垃圾收集上花费多少时间（时间比例），堆的大小会依据该目标动态调整	通常作为 Throughput 收集器调优的第一步，Throughput 收集器计算出的默认堆大小如果无法达到应用的目标就会对其进行调整	“堆大小的自适应调整和静态调整”，P105
-XX:-AggressiveHeap	对于配置了大量内存的机器，如果只运行单一的虚拟机，并为该虚拟机分配了大容量的堆，使用该标志可以开启一系列的优化标志	不推荐使用该标志，理想的情况是尽可能地按照需要使用具体的调优标志	“AggressiveHeap 标志”，P136

表A-5：CMS收集器调优标志

标志	功能	使用场景	更多信息
-XX:CMSInitiatingOccupancyFraction=N	设定何时 CMS 收集器可以开启老年代空间的后台扫描	CMS 收集器如果发生并发模式失效，就应该适当减少该参数的值	“理解 CMS 收集器”，P109

（续）

标志	功能	使用场景	更多信息
-XX:+UseCMSInitiatingOccupancyOnly	通知 CMS 收集器仅通过 -XX:CMSInitiatingOccupancyFraction 标志决定何时启动 CMS 后台扫描	与 CMSInitiatingOccupancyFraction 标志一起使用	“理解 CMS 收集器”，P109
-XX:ConcGCThreads=*N*	设定 CMS 收集器进行后台扫描使用的线程数	CMS 收集器发生并发模式失效，同时又有大量的空闲 CPU 资源时，可以通过该标志增加扫描线程数	“理解 CMS 收集器”，P109
-XX:+CMSPermGenSweepingEnabled	通知 CMS 收集器对永久代进行整理	采用 CMS 收集器的应用服务器进行了大量的类卸载时	“理解 CMS 收集器”，P109
-XX:CMSInitiatingPermOccupancyFraction=*N*	设定何时 CMS 收集器可以对永久代进行扫描	CMSPermGenSweepingEnAbled 标志开启的同时发生了由于永久代的回收不够快导致的 Full GC 时	“理解 CMS 收集器”，P109
-XX:+CMSClassUnloadingEnabled	设定 CMS 收集器在永久代扫描结束后卸载类	与 CMSPermGenSweepingEnabled 结合使用	“理解 CMS 收集器”，P109
-XX:+CMSIncrementalMode	以增量模式运行 CMS 收集	如果 CPU 资源有限，不过你仍然需要使用 CMS 收集器时	“增量式 CMS 垃圾收集”，P117
-XX:CMSIncrementalModeSafetyFactor=*N*	该参数会影响增量式 CMS 垃圾收集后台线程的运行频度	采用增量式 CMS 收集，同时又发生了并发模式失效，此时应该减小该参数	“增量式 CMS 垃圾收集”，P117
-XX:CMSIncrementalDutyCycleMin=*N*	该参数会影响增量式 CMS 垃圾收集后台线程运行的频度	采用增量式 CMS 垃圾收集，同时又发生了并发模式失效时，可以增大该参数进行调节，不过推荐的方式还是使用 CMSIncrementalModeSafetyFactor	“增量式 CMS 垃圾收集”，P117
-XX:CMSIncrementalDutyCycleMax=*N*	该参数会影响增量式 CMS 垃圾收集后台线程的运行频度	采用增量式 CMS 垃圾收集，同时又发生了并发模式失效时，可以增大该参数进行调节，不过推荐的方式还是使用 CMSIncrementalModeSafetyFactor	“增量式 CMS 垃圾收集”，P117
-XX:+CMSIncrementalDutyCycle	该参数会影响增量式 CMS 垃圾收集后台线程的运行频度	使用 CMSIncrementalDutyCycleMin 时，你需要设置这个标志	“增量式 CMS 垃圾收集”，P117

表A-6：G1收集器的调优标志

标志	功能	使用场景	更多信息
-XX:MaxGCPauseMillis=*N*	设置 G1 收集的最长停顿时间；G1 收集算法会依据该参数进行相应的调整	是 G1 收集器调优的第一步，可以尝试调整（增大）这个值以避免发生 Full GC	“理解 G1 垃圾收集器”，P118
-XX:ConcGCThreads=*N*	设置 G1 收集后台扫描的线程数	有足够的 CPU 资源，同时又遭遇并发模式失效时，可以考虑调整该参数	“理解 G1 垃圾收集器”，P118
-XX:InitiatingHeapOccupancyPercent=*N*	设置何时 G1 收集开启后台扫描	如果 G1 收集发生并发模式失效，应该减小这个参数值	“理解 G1 垃圾收集器”，P118
-XX:G1MixedGCCountTarget=*N*	G1 收集器尝试回收老年代分区的垃圾时，要设置混合 GC 的次数	如果 G1 收集器发生了并发模式失效，减小这个值；如果混合式 GC 的周期过长，增大这个值	“理解 G1 垃圾收集器”，P118
-XX:G1HeapRegionSize=*N*	设置 G1 分区的大小	对非常大的堆，或者应用需要分配非常巨大的对象时，可以增大这个值进行调节	“G1 分区的大小”，P133

表A-7：内存管理标志

标志	功能	使用场景	更多信息
-XX:+HeapDumpOnOutOfMemoryError	JVM 抛出一个内存异常就自动生成一个堆转储	如果应用抛出堆空间或者永久代引发的内存错误可以考虑开启这个标志，分析堆上的内存泄漏	“内存溢出错误”，P146
-XX:HeapDumpPath=<path>	设置自动堆转储保存的文件名	由于内存错误或者 GC 事件引起的堆转储，可以通过这个标志指定保存路径，不再是 java_pid.hprof（假设所有这些标志都已经开启）	“内存溢出错误”，P146
-XX:SoftRefLRUPolicyMSPerMB=N	控制使用多长时间后软引用对象可以被回收	内存不足的情况下，减小这个值可以帮助更快地回收软引用对象	“弱引用、软引用与其他引用”，P165
-XX:MaxDirectMemorySize=N	设置通过 ByteBuffer 类的 `allocateDirect()` 方法可以分配多少本地内存	如果你希望限制程序能够分配的直接内存量，可以考虑使用该标志。应注意的是，分配超过 64 MB 直接内存的对象不需要设置这个标志	“内存占用”，P176
-XX:+UseLargePages	设置 JVM，在可行的前提下，从操作系统的大页面系统中分配页面	如果操作系统支持，使用这个标志通常都能改善性能	“大页”，P182

（续）

标志	功能	使用场景	更多信息
-XX:+LargePageSizeInBytes=*N*	设置JVM分配指定大小的大页面（仅适用于Solaris）	在Solaris系统上，增大这个值（譬如调整到256 MB）通常能取得较好的性能	“大页”，P182
-XX:+StringTableSize=*N*	设置JVM用于保存内部字符串中Hashtable的大小	如果应用需要临时保存大量的字符串，可以考虑增大这个值	“字符串的保留”，P157
-XX:+UseCompressedOops	为对象引用模拟35位长的指针	这个标志对于堆的容量小于32 GB的系统默认开启，禁用该标志不会带来性能的提升	“压缩的OOP”，P185
-XX:+PrintTLAB	在垃圾收集日志中输出TLAB相关的概略信息	如果你使用的JVM不支持JFR，通过这个标志可以确保TLAB的分配是有效的	TLAB，P129
-XX:TLABSize=*N*	设置TLAB的大小	如果应用正在进行大量的TLAB外分配，可以通过这个标志增大TLAB的容量	TLAB，P129
-XX:-ResizeTLAB	关闭TLAB的大小调整功能	一旦设定好TLAB的大小，请确保关闭这个标志	TLAB，P129

表A-8：线程处理标志

标志	功能	使用场景	更多信息
-Xss<N>	设置线程的本地堆栈大小	尤其是在32位Java虚拟机上，减小这个值能为JVM的其他部分留出更多的可用内存	“调节线程栈大小”，P211
-XX:-BiasedLocking	关闭Java虚拟机的偏向锁算法	对基于线程池的应用，这个标志常常能帮助提升性能	“偏向锁”，P212

表A-9：杂项JVM调优标志

标志	功能	使用场景	更多信息
-XX:+AlwaysLockClassLoader	将类加载方案由Java 7的并行加载回退到Java 6的非并行加载	大型系统上，类的载入通常由单一线程完成，使用这个标志对启动性能有一定的提升	“类加载”，P280
-XX:-StackTraceInThrowable	关闭发生异常时，堆栈跟踪信息的收集	堆栈调用非常深的系统上，经常会抛出异常（这些场景中，修改代码，抛出更少的异常机会非常小），可以考虑使用该标志	“异常”，P287
-XX:-RestrictContended	允许非JDK代码使用@Contended注解	如果应用程序代码使用了@contended注释方式填充变量，为避免伪共享，需要设置这个标志	“Contended注解”，P210

（续）

标志	功能	使用场景	更多信息
-XX:-EnableContended	禁用 JDK 代码中的 @Contended 注解	推荐开启该标志。不过在某些 JDK 类中，禁用该标志可以节省少量空间	“Contended 注解”，P210
-XX:+AggressiveOpts	启用某些尝鲜型的 JVM 优化。这些优化在将来的发布版本中可能会变成默认设置	你可以使用这个标志开启这些试用性质的优化进行测试，但是也要意识到这些标志随着新的 JVM 发布，行为可能会发生变化	“AggressiveOpts 标志”，P296

表A-10：Java飞行记录器控制标志

标志	功能	使用场景	更多信息
-XX:+FlightRecorder	启用 Java 飞行记录器	推荐开启飞行记录器，它的开销非常小，只有在实际记录发生时有少许的开销（这时的开销多少，依据使用的特性不同，会有所不同，不过总体而言还是相当小的）	“Java 飞行记录器”，P46
-XX:+FlightRecorderOptions	通过这个命令行标志可以设置默认的记录标志	控制 JVM 默认进行哪些数据的记录	“Java 飞行记录器”，P46
-XX:+UnlockCommercialFeatures	允许 JVM 使用付费（非开源）功能	如果你购买了对应的许可，通过设置该标志可以启用 Java 飞行记录器的付费功能	“Java 飞行记录器”，P46

作者简介

Scott Oaks 是 Oracle 公司的一位架构师，专注研究 Oracle 中间件软件的性能。加入 Oracle 之前，他曾于 Sun Microsystem 公司任职多年，在多个技术领域都有建树，包括 SunOS 的内核、网络程序设计、Windows 系统的远程方法调用（RPC）以及 OPEN LOOK 虚拟窗口管理器。1996 年，Scott 成为 Sun 公司的 Java 布道师，并于 2001 年加入 Sun 公司的 Java 性能小组——从那时起他就一直专注于 Java 的性能提升。此外，Scott 也在 O'Reilly 出版社出版了多部书籍，包括 *Java Security*、*Java Threads*、*JXTA in a Nutshell* 和 *Jini in a Nutshell*。

关于封面

本书封面上的动物是高鼻羚羊（也叫赛加羚羊），通常我们称之为赛加羚。它们最独特的地方是那硕大而灵活的鼻子，有一部分甚至覆于其嘴部之上。高鼻羚羊的鼻子能够在夏天帮助过滤砂尘，而在冬天又能使吸入的空气在进入肺腔之前加热。

高鼻羚羊的皮毛能随着季节自动进行调节：寒冷的季节，毛白且厚实；炎热的季节，毛色会转变为肉桂色，并且厚度也显著变薄。高鼻羚羊的肩高大约为 2 至 3 英尺，体重在 80~140 磅之间。高鼻羚羊有细而长的腿，在逃避猎食者时的奔跑速度可高达每小时 80 英里。它们栖息于半沙漠化的草原地区，那里生长着各种各样的野草和灌木（有些灌木甚至只适合高鼻羚羊食用，对其他动物却有毒性），高鼻羚羊就依赖这些植物为生。

高鼻羚羊通常喜群居，一般 30 至 40 头高鼻羚羊聚居为生。不过到冬天迁移的时节，它们会汇集成上千头的大集群一起行动。它们的交配期始于十一月末，也就是整个群落向南完成迁徙之后。在这期间，雄性高鼻羚羊吃得很少，相互之间会进行激烈的争斗，最终有大量的高鼻羚羊力竭而死。角逐的获胜者和多头雌性高鼻羚羊交配。次年的四月后期，每只雌性高鼻羚羊会产下一到两只幼崽。

高鼻羚羊几乎全部栖居于哈萨克斯坦，只有一小部分隔绝的分支生活在蒙古。虽然它们曾经广泛地分布于中亚的草原，但由于过去几十年的大量猎捕，它们现在已经濒临灭绝，被列为“濒危保护动物”。苏联的解体放纵了盗猎者们，无节制的猎杀导致高鼻羚羊种群的数量骤降 97%。种群的恢复困难重重，因为只有雄性高鼻羚羊有角，而高鼻羚羊角是名贵的中药材，需求旺盛。受利益的驱使，盗猎者们几乎杀尽了所有的雄性高鼻羚羊，严重损害了这个种群的繁衍能力。

封面图片取自一个活动板，来源未知。

O'REILLY®

Java性能权威指南

市面上介绍Java的书有很多，但专注于Java性能的并不多，能游刃有余地展示性能优化难点的更是凤毛麟角，本书即是这样一本十分珍贵的指南。通过使用JVM和Java平台，以及Java语言和应用程序接口，本书详尽讲解了Java性能调优的相关知识，帮助读者深入理解这一主题，最终使程序如虎添翼。

通过阅读本书，你可以：

- **运用四个基本原则最大程度地提升性能测试的效果**
- **使用JDK中自带的工具收集Java应用的性能数据**
- **理解JIT编译器的优缺点**
- **调优JVM垃圾收集器以减少对程序的影响**
- **学习管理堆内存和JVM原生内存的方法**
- **了解如何最大程度地优化Java线程及同步的性能**
- **解决Java EE和Java SE应用程序接口的性能问题**
- **改善Java驱动的数据库应用程序的性能**

Scott Oaks是Oracle公司的架构师，专注于Oracle中间件软件的性能提升。他自1987年起服务于Sun公司，后成为Sun公司的Java布道师，并于2001年加入Sun公司的Java性能小组，其后专注于Java性能工作。他著有一系列涉及Java安全、Java线程及Jini等内容的图书。

“Scott在这本书里深入介绍了JIT编译器、垃圾收集算法、线程同步等复杂问题，讲解了很多应用性能分析以及优化的方法，让我受益匪浅。推荐给所有想了解性能优化的Java程序员。”

——Thorben Janssen
资深Java程序员，
独立Java培训师，
JSR 365专家组成员

“这是我今年读到的所有Java书中最有用的一本。作者全方位讲解了Java应用程序的方方面面，下至核心术语、调优工具等基础知识，上至Java EE、Java 8等进阶话题，环环相扣。每一位急于了解Java性能调优或JVM工作方式的开发者，都必备此书。”

——亚马逊读者评论

PROGRAMMING / JAVA

封面设计：Karen Montgomery 张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机 / 程序设计 / Java

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-41376-5

ISBN 978-7-115-41376-5
定价：79.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks